EVOLUTION OF WEB-BASED SYSTEMS IN MODEL DRIVEN ARCHITECTURE

PhD Thesis

Bing Qiao

A thesis submitted to in fulfilment of the requirements for the degree of

Doctor of Philosophy

De Montfort University

February, 2006

Abstract

The complexity and size of commercial Web-based systems present a grand challenge to the traditional methodology of software evolution. However, compared to the huge advance of software development technology over the last two decades, the progress of software evolution research and practice, especially for Web-based systems, is still very limited.

Modern software development is built on a number of principles, paradigms, and tools. Those building blocks provide a standard, flexible and integrated way to develop and deliver a definite product.

From programming language to operating system, from Integrated Development Environment (IDE) to software process model, many alternatives can coexist and be regarded as "standards" due to their popularity or authority. To build a commercial web application, it is completely up to the development team to choose the operating system, programming language, IDE and development process. Thanks to standards built on techniques such as XML and UML, those building blocks could be integrated seamlessly and flexibly no matter how or by whom they were created. Finally, regardless of the technology adopted for development, the product of any software development should be always a working system, an instantiation of the requirement specification.

However, when it comes to software evolution, there is no standard, flexible and integrated way to evolve and deliver a definite product. The booming development of Web related technologies only complicates the situation. This research presents a unified solution to Web-based system evolution, which consists of three components: Web-based systems understanding, Web-based systems representation and evolvable Web Application Framework:

- Web-based systems understanding. A successful evolution of a legacy system relies on an appropriate understanding of its functionality, context and architecture. Traditionally software reverse engineering techniques, either formal or cognitive, have been used for this purpose. This research presents a

unified method for understanding Web-based systems, where a formal method and a data mining technique are developed to decipher program logic and the relationships between different components.

- Web-based systems representation. The information hidden in Web-based systems can be divided into five categories: source code of control logic, source code of presentation logic, configuration information, input/output files and data model. Each category of the information has its value in evolving the related Web-based system. While source code and configuration files are vital for understanding the whole system, the input/output files and data model determine the flexibility for maintenance and future development. To effectively manipulate the information hidden in Web-based systems, representations for each of those categories are defined in this research.

- Evolvable Web Application Framework. Most Web-based systems are built upon a certain software infrastructure. An infrastructure provides services such as resource pooling, thread management, service lookup and data access layer. This research will look at existing frameworks and develop an alternative infrastructure that we believe is essential to successful evolution of Web-based systems.

A comprehensive case study will be given to evaluate the proposed solution in different aspects. Conclusion is drawn based on analysis, which verifies the feasibility of the proposed solution. Further research areas are also discussed.

Table of Contents

EVOLUTION OF WEB-BASED SYSTEMS IN MODEL DRIVEN ARCHITECTURE 1

ABSTRACT I

TABLE OF CONTENTS III

TABLE OF FIGURES X

TABLE OF TABLES XII

TABLE OF LISTINGS XIII

ACKNOWLEDGEMENTS XVI

DECLARATION XVII

GLOSSARY XVIII

ACRONYMS XX

CHAPTER 1 INTRODUCTION 1

1.1 SOFTWARE EVOLUTION: OBSTACLES 1

1.2 SOFTWARE EVOLUTION: WEB 3

1.2.1 Issues about the Users 3

1.2.2 Issues about the Developers 4

1.2.3 Issues about Software Engineering 4

1.2.4 Issues about Web Application Frameworks 4

1.3 RESEARCH METHOD 5

1.3.1 Observation 6

1.3.2 Experiment 8

1.3.3 Analyses 8

1.4 MAJOR CONTRIBUTIONS 9

1.5 SUCCESS CRITERIA 11

1.6 THESIS ORGANISATION 11

CHAPTER 2 BACKGROUND 13

2.1 SOFTWARE EVOLUTION 13

2.1.1 Software Reengineering vs. Software Maintenance 14

2.1.2 Legacy Systems: Obsolete Techniques vs. Poor Development Practices 17

2.1.3 Reverse Engineering: Formal vs. Cognitive 17

2.2 PATTERNS IN SOFTWARE EVOLUTION 21

2.2.1 Pattern Form *21*
2.2.2 Pattern Language *22*
2.2.3 Pattern Complexity *22*
2.3 EVOLUTION OF WEB DEVELOPMENT 23
2.3.1 Servlet-based Architecture *23*
2.3.2 JSP Model 1 Architecture *24*
2.3.3 JSP Model 2 Architecture *24*
2.4 XML AND UML IN SOFTWARE EVOLUTION 25
2.4.1 XML for Information Exchange *26*
2.4.2 XML for Representation *27*
2.4.3 UML for Representation *28*
2.4.4 UML for Automation *29*
2.5 MODEL DRIVEN ARCHITECTURE 32
2.6 MODELING WEB-BASED SYSTEMS 33
2.6.1 Web Modeling Language *34*
2.6.2 Web Application Extension *35*
2.7 SUMMARY 36

CHAPTER 3 RELATED RESEARCH 38

3.1 SOFTWARE REENGINEERING 38
3.1.1 Formal Methods for Reverse Engineering *38*
3.1.2 Software Architecture Recovery *42*
3.1.3 Web-based Systems Evolution *45*
3.2 ARCHITECTURE DESCRIPTION LANGUAGE 46
3.3 MODEL DRIVEN FRAMEWORK 47
3.3.1 OptimalJ *47*
3.3.2 Arcstyler *48*
3.4 WEB APPLICATION FRAMEWORK 49
3.4.1 Struts *49*
3.4.2 WebWork *50*
3.5 SUMMARY 50

CHAPTER 4 A PROPOSED APPROACH TO WEB-BASED SYSTEMS EVOLUTION 52

4.1 SOFTWARE EVOLUTION: ROAD TO EVOLVABLE WEB-BASED SYSTEMS 52
4.1.1 Round-Trip Engineering with MDA *55*
4.1.2 Test Driven Development *58*
4.1.3 Adaptable MDA *58*
4.2 SOFTWARE EVOLUTION PROCESS 60
4.2.1 Concept and Use Case Models *62*

4.2.2 Program Translation *63*
4.2.3 Modules Identification *64*
4.2.4 Components Identification *67*
4.2.5 Architecture Recovery *70*
4.2.6 MDA Models Identification *73*
4.2.7 Architecture Migration *75*
4.3 SUMMARY 77

CHAPTER 5 WEB APPLICATION INFRASTRUCTURE AND FRAMEWORK FOR EVOLVABLE WEB-BASED SYSTEMS **79**

5.1 DESIGN ISSUES OF EVOLVABLE APPLICATIONS 79
5.1.1 Inheritance and Interface in OO *79*
5.1.2 Design Patterns for Building Evolvable Applications *80*
5.1.3 Application Registry for Building Evolvable Applications *81*
5.2 WEB APPLICATION INFRASTRUCTURE 82
5.2.1 Design Issues of Web Application Infrastructure *82*
5.2.2 Evolvable Web Application Infrastructure *84*
5.3 WEB APPLICATION FRAMEWORK 89
5.3.1 Design Issues of Web Application Framework *89*
5.3.2 Model, View and Controller for Evolvable Web Framework *91*
5.3.3 Evolvable Web Application Framework *93*
5.4 SUMMARY 97

CHAPTER 6 A FORMAL METHOD FOR HIGH-LEVEL SPECIFICATIONS EXTRACTION . 99

6.1 SPECIFICATION EXTRACTION FRAMEWORK IN MDA 100
6.2 REQUIREMENTS FOR SPECIFICATION EXTRACTION RULE 101
6.3 NOTATIONS FOR SPECIFICATION EXTRACTION RULES 103
6.4 JAVA/PSL TRANSLATION RULES 105
6.5 ELEMENTARY ABSTRACTION RULES 107
6.6 ARCHITECTURE ABSTRACTION RULES 111
6.7 JAVA-BASED COMPILER FOR SPECIFICATION EXTRACTION RULES 111
6.8 SUMMARY 116

CHAPTER 7 A COGNITIVE METHOD FOR ARCHITECTURE RECOVERY 118

7.1 RELATIONSHIP ANALYSIS PRINCIPLES 120
7.1.1 Generalisation Relationship *120*
7.1.2 Aggregation Relationship *120*
7.1.3 Classification Relationship *121*
7.1.4 Association Relationship *121*
7.2 RELATIONSHIPS IN OBJECT-ORIENTED SYSTEMS 123

7.2.1 Quantification of Relationships *124*
7.2.2 Unification of Relationships *127*
7.2.3 Clustering of Relationships *128*
7.2.4 Hierarchical Clustering *128*
7.3 RELATIONSHIPS IN WEB-BASED SYSTEMS 130
7.4 ARCHITECTURE DESCRIPTION 131
7.4.1 Architecture Representation in UML *133*
7.4.2 Architecture Description in XML *135*
7.5 SUMMARY 141

CHAPTER 8 PATTERNS FOR MODELS IDENTIFICATION IN WEB-BASED SYSTEMS EVOLUTION 142

8.1 DOMAIN ARCHITECTURAL PATTERNS 142
8.1.1 Management Information Systems *142*
8.1.2 Resource Allocation and Tracking systems *144*
8.1.3 Manufacturing Systems *145*
8.1.4 Access Control Systems *145*
8.1.5 Lifecycle Model *147*
8.2 SYSTEM DESIGN PATTERNS 147
8.2.1 Web Presentation Patterns *147*
8.2.2 Distributed Computing: Instance-based Patterns *152*
8.2.3 Distributed Systems: Service-based Patterns *155*
8.3 INTEGRATION PATTERNS 156
8.3.1 File Transfer *158*
8.3.2 Shared Database *159*
8.3.3 Remote Procedure Invocation *159*
8.3.4 Messaging *160*
8.4 SUMMARY 161

CHAPTER 9 XML TRANSFORMERS FOR EVOLVABLE DATA MANAGEMENT 163

9.1 XML-BASED DATA MANAGEMENT 163
9.1.1 Standardisation *164*
9.1.2 Self-Descriptiveness *164*
9.1.3 Separation of Concerns *166*
9.1.4 Interoperability *167*
9.1.5 Flexibility *168*
9.1.6 Reusability *169*
9.1.7 Extensibility *170*
9.1.8 Hierarchical Composition *172*

9.1.9 Internationalisation *173*
9.1.10 Supporting Tools *173*
9.1.11 Highly Adapted for Web-based Systems *174*
9.2 XML INTEGRATION 174
9.2.1 XML-based APIs *174*
9.2.2 Aggregation *175*
9.2.3 Business to Business Integration *175*
9.2.4 Information Exchange and Distribution *175*
9.2.5 Web Services *176*
9.3 BENEFITS OF USING XML IN SOFTWARE EVOLUTION 178
9.4 XML TRANSFORMATION 181
9.5 DESIGN AND GRAMMAR 182
9.5.1 Instance Document Design *182*
9.5.2 Grammar Description Document Design *182*
9.5.3 Schemas for Grammar Description Documents *183*
9.5.4 Schemas for Instance Documents *184*
9.6 ARCHITECTURE OF XML-AWARE EVOLUTION 184
9.6.1 Source Transformer Processing *185*
9.6.2 Target Transformer Processing *186*
9.7 SUMMARY 186

CHAPTER 10 XML TRANSFORMATION TOOL 187

10.1 CONTROLLER STRUCTURE 189
10.2 TRANSFORMERBASE CLASS 191
10.2.1 Constructor *191*
10.2.2 loadGrammarDocument *191*
10.3 SOURCETRANSFORMERBASE CLASS 192
10.3.1 Constructor *192*
10.3.2 writeDocument *192*
10.3.3 abstract processSourceFile *193*
10.4 TARGETTRANSFORMERBASE CLASS 193
10.4.1 Constructor *193*
10.4.2 abstract processTargetDocument *193*
10.5 RECORDHANDLERBASE CLASS 194
10.5.1 Constructor *194*
10.5.2 createDataUnit *195*
10.5.3 getElementTextNode *195*
10.5.4 getFieldValue *195*
10.5.5 getDelimiter *195*

10.5.6 setRecordTerminator *195*
10.6 RECORDREADERBASE CLASS 195
10.6.1 Constructor *196*
10.6.2 readRecordVariableLength *196*
10.6.3 setOutputDocument *196*
10.6.4 toXMLType *197*
10.6.5 writeLegacyRecordToXML *197*
10.6.6 abstract getRecordType *197*
10.6.7 abstract parseLegacyRecord *197*
10.6.8 abstract readLegacyRecord *197*
10.7 RECORDWRITERBASE CLASS 197
10.7.1 Constructor *198*
10.7.2 parseXMLRecord *198*
10.7.3 abstract writeXMLRecordToLegacy *198*
10.8 DATAUNITBASE CLASS 198
10.8.1 Constructor *199*
10.8.2 getField *199*
10.8.3 putByte *199*
10.8.4 putField *200*
10.8.5 toElement *200*
10.8.6 convert2Legacy *200*
10.8.7 prepareOutput *200*
10.8.8 convert2XML *200*

CHAPTER 11 CASE STUDY 201

11.1 ABSTRACTION AND REPRESENTATION: TIC TAC TOE 201
11.1.1 Abstraction *202*
11.1.2 Representation *206*
11.2 LEGACY SYSTEMS EVOLUTION: WEB-BASED XML REPORTING SOLUTION 206
11.2.1 Input of CSV to XML: Invoice *206*
11.2.2 Input of XML to CSV: Purchase Order *209*
11.2.3 Grammar Description and Schema *213*
11.2.4 Output of CSV to XML *214*
11.2.5 Output of XML to CSV *218*
11.2.6 Web-based XML Distribution *219*
11.3 PROPOSED WEB APPLICATION INFRASTRUCTURE AND FRAMEWORK: PETSTORE ONLINE 219
11.3.1 Business Logic Implementation *220*
11.3.2 Web Tier Implementation *223*
11.4 SUMMARY 226

CHAPTER 12 CONCLUSION **228**

12.1 SUMMARY OF THESIS 229

12.2 SUCCESS CRITERIA REVISITED 231

12.3 APPLICABILITY OF PROPOSED APPROACH 234

12.4 LIMITATION AND FUTURE WORK 235

REFERENCES **237**

APPENDIX A UML PROFILE FOR WEB APPLICATION EXTENSIONS **248**

APPENDIX B GRAMMAR DESCRIPTION SCHEMAS **250**

B.1 CSV SOURCE GRAMMAR DESCRIPTION SCHEMA 250

B.2 CSV TARGET GRAMMAR DESCRIPTION SCHEMA 250

B.3 CSV GRAMMAR DESCRIPTION COMMON SCHEMA 251

APPENDIX C WEB APPLICATION INFRASTRUCTURE AND FRAMEWORK **254**

C.1 WEB APPLICATION INFRASTRUCTURE 254

C.1.1 Design Issues of Evolvable Applications *254*

C.1.2 Enhanced Bean-based Manipulation *257*

C.2 WEB APPLICATION FRAMEWORK 259

C.2.1 A Basic Controller Implementation *259*

C.2.2 A Controller Exposing Bean Properties *260*

C.2.3 A Method Mapping Controller *261*

APPENDIX D RELATIONSHIP ANALYSIS OF WEB-BASED SYSTEMS **263**

D.1 RELATIONSHIPS IN STRUTS CONFIGURATION FILES 263

D.1.1 Struts Configuration File *263*

D.1.2 Relationships in Struts Configuration File *266*

D.2 RELATIONSHIPS IN iBATIS CONFIGURATION FILES 270

D.2.1 iBATIS Configuration Files *270*

D.2.2 Relationships in iBATIS Configuration File *271*

APPENDIX E PUBLICATIONS **274**

Table of Figures

Figure 2-5-1 Evolvable Systems in B2B and J2EE 27
Figure 2-5-2 Pipe and Filter Pattern 28
Figure 2-5-3 Client-server Architecture 29
Figure 2-6-1 MDA process 33
Figure 4-1-2. Reengineering within MDA Environment 57
Figure 4-2-1 Cubic Model for Software Reengineering 60
Figure 4-2-2 Unified Software Reengineering Process 61
Figure 4-2-3 Process of Program Translation 64
Figure 4-2-4 Process of Modules Identification 66
Figure 4-2-5 Process of Components Identification 68
Figure 4-2-6 Process of Architecture Recovery 72
Figure 4-2-7 Process of Model Identification 75
Figure 4-2-8 Process of Architecture Migration 76
Figure 5-2-1 Hierarchy of Evolvable Containers 85
Figure 6-1-1 Specification Extraction Framework 100
Figure 6-7-1 Simplified Compiler for PSL 113
Figure 7-1-1 Hierarchy of Relationships 119
Figure 7-2-1 Representation of Relationships 123
Figure 7-2-2 Process of Agglomerative Clustering 130
Figure 7-4-1 UML Diagram for Component 134
Figure 7-4-2 UML Diagram for Connector 135
Figure 7-4-3 UML Diagram for Configuration 135
Figure 7-4-4 UML Diagram for Component & Connector 135
Figure 8-1-1 MIS Domain Pattern 143
Figure 8-1-2 RAT Domain Pattern 144
Figure 8-1-3 MAN Domain Pattern 145
Figure 8-1-4 ACS Domain Pattern 146
Figure 8-2-1 MVC System Design Pattern 148
Figure 8-2-2 Page Controller System Design Pattern 150
Figure 8-2-3 Front Controller System Design Pattern 151
Figure 8-2-4 Intercepting Filter System Design Pattern 152
Figure 8-2-5 Broker System Design Pattern 153
Figure 8-2-6 Singleton System Design Pattern 154
Figure 8-2-7 Service Interface System Design Pattern 155
Figure 8-3-1 File Transfer Integration Pattern 158
Figure 8-3-2 Shared Database Integration Pattern 159

Figure 8-3-3 RPI Integration Pattern 160
Figure 8-3-4 Messaging Integration Pattern 161
Figure 9-1-1 Data Reuse for Payment Amount. 170
Figure 9-1-2 Extended XML Transaction File 171
Figure 9-2-1 Web services Integration 177
Figure 9-3-1 System flow chart before reengineering 179
Figure 9-3-2 XSLT Transformation for System Evolution 179
Figure 9-3-3 Surf-and-seek Portal-based Access 180
Figure 9-3-4 Focused, Habitual, Portal-based Access 180
Figure 9-6-1 Source Transformer Collaboration Diagram 185
Figure 9-6-2 Target Transformer Collaboration Diagram 186
Figure 10-1-1 Source CSV File 187
Figure 10-1-2 Grammar Description File and Transformation Process 188
Figure 10-1-3 Resulting XML Documents 188
Figure 10-2-1 TransformerBase 191
Figure 10-3-1 SourceTransformerBase 192
Figure 10-4-1 TargetTransformerBase 193
Figure 10-5-1 RecordHandlerBase 194
Figure 10-6-1 RecordReaderBase 196
Figure 10-7-1 RecordWriterBase 198
Figure 10-8-1 DataUnitBase 199
Figure 11-1-1 Noughts and Crosses 201
Figure 11-1-2 Architecture Description of Naugts and Crosses 206
Figure 11-2-11 CSV to RSS Feed 219

Table of Tables

Table 4-1-1 AMDA and TDD comparison 60
Table 6-4-1 Mapping between Java and PSL Types 106
Table 6-6-1 Architecture Abstraction Rules 111
Table 9-2-1 Web services Integration 177
Table 9-5-1 Elements of Grammar Description Document 183
Table 11-2-1 Logical Layout for the Invoice 207
Table 11-2-2 Logical Layout for the Purchase Order 210
Table A-1 UML Profile for Web Application Extensions 249
Table D-1-1 Relationships in Struts Configuration File 270
Table D-2-1 Relationships in iBatis Configuration File 273

Table of Listings

Listing 5-1-1 Implementation of Singleton in Java 81
Listing 5-1-2 Registry Object for Evolvable Application 82
Listing 5-2-1 Setter Dependency Push 87
Listing 5-2-2 Constructor Dependency Push 87
Listing 6-3-1 Notations for Specification Extraction Rules 103
Listing 6-4-1 Translation between Java Class and PSL Class 105
Listing 6-4-2 Translation between Java while Statement and PSL while Statement 106
Listing 6-5-1 Reflexive Abstraction Rule 107
Listing 6-5-2 Transitive Abstraction Rule 107
Listing 6-5-3 Monotonic Abstraction Rule 108
Listing 6-5-4 Weakening Abstraction Rule 109
Listing 6-5-5 Attribute Abstraction Rule 109
Listing 6-5-6 Sequence Folding Abstraction Rule 110
Listing 6-7-1 Grammar Rules for PSL 112
Listing 6-7-2 Class AbstractConstruct 115
Listing 6-7-3 Class Context 115
Listing 6-7-4 Class NonTerminalNode 115
Listing 6-7-5 Class Visitor 116
Listing 6-7-6 Class SequenceFoldingVisitor 116
Listing 7-4-1 XML Schema for Component 137
Listing 7-4-2 XML Schema for Connector 137
Listing 7-4-3 XML Schema for Configuration 139
Listing 7-4-4 XML Schema for M&R Database 140
Listing 7-4-5 XML Schema for C&C Database 140
Listing 9-1-1 Customer Information Document 165
Listing 10-1-1 Controller for Legacy to XML Transformation 190
Listing 10-1-2 Controller for XML to Legacy Transformation 191
Listing 11-1-1 Source Code of the Server Side Class XOServer 203
Listing 11-1-2 PSL Code of the Server Side Class XOServer 204
Listing 11-1-3 XOServer After Appying Elementary Abstraction Rules 204
Listing 11-1-4 XOServer After Applying Weakening Abstraction Rules 205
Listing 11-2-1 Input CSV File (Invoices.csv) 208
Listing 11-2-2 Invoice Schema (CSVInvoice.xsd) 209
Listing 11-2-3 Purchase Order 1 211

Listing 11-2-4 Purchase Order 2 212
Listing 11-2-5 Purchase Order 3 212
Listing 11-2-6 Purchase Order Schema (CSVPurchaseOrder.xsd) 213
Listing 11-2-7 CSV File Grammar 213
Listing 11-2-8 CSV Row Grammar 214
Listing 11-2-9 Output Documents 218
Listing 11-2-10 Purchase Order 1 219
Listing 11-3-1 Inventory Interface 220
Listing 11-3-2 OrderProcessor Interface 220
Listing 11-3-3 Bean Definitions from Application Registry 221
Listing 11-3-4 OrderProcessor Bean Definition 222
Listing 11-3-5 Bean Definition for petController 222
Listing 11-3-6 JavaBean Setters for Controller Configuration 223
Listing 11-3-7 PetController Definition 224
Listing 11-3-8 ControllerMapping Definition 224
Listing 11-3-9 MethodMapper Definition 225
Listing 11-3-10 Request Handling 225
Listing 11-3-11 Bean Properties Initialised from XML Configuration File 225
Listing 11-3-12 Showing an Existing Order 226
Listing 11-3-13 Creating a New Order 226
Listing B-1 CSVSourceGrammarDescription.xsd 250
Listing B-2 CSVTargetGrammarDescription.xsd 251
Listing B-3 CSVCommonGrammarDescription.xsd 253
Listing C-1-1 Example of Template Method 254
Listing C-1-2 Template Methods for Workflow Steps 255
Listing C-1-3 Interface for Strategy Design Pattern 255
Listing C-1-4 Bean Property for Setting Helper Object 255
Listing C-1-5 Example of Strategy Design Pattern 256
Listing C-1-6 RowCallbackHandler Interface 256
Listing C-1-7 Callback Pattern to Hide Implemenation Details 257
Listing C-1-8 Observer Design Pattern in GUI Component 257
Listing C-1-9 TimeBean Definition 258
Listing C-1-10 Usage of BeanHandler 259
Listing C-2-1 A Simple Controller Implementation 259
Listing C-2-2 A Simple Controller with Exposed Beans 260
Listing C-2-3 Configuration of view Object 261
Listing C-2-4 Request Handling Method Signature 262
Listing C-2-5 MethodMapper Interface 262
Listing D-1-1 Struts Configuration File for Pet Store 266

Listing D-2-1 Item.xml 271

Acknowledgements

I wish to acknowledge the financial support from De Montfort University (DMU) for this three-year research work.

I wish to express my most sincere thanks to my supervisor Prof. Hongji Yang for his invaluable advice, support and encouragement. I will carry on his guidance through out my life.

I appreciate the valuable feedback provided by my thesis readers, Professor Kecheng Liu and Dr Antonio Cau. I am grateful to Dr. Dan Zhao for providing me the opportunity to learn and practice at Fimat International Banque, where I was able to connect academic research with industrial development.

It is fortunate for me to work and study with the amazing members of Software Technology Research Laboratory (STRL). In particular, I would like to thank Shaoyun Li and Jianzhi Li for all their help and encouragement.

This thesis would not have been possible without the continuous support of my wife Yue Wang and the rest of my family who gave me the strength and will to succeed.

Declaration

I declare that the work described in this thesis was originally carried out by me during the period of registration for the degree of Doctor of Philosophy at De Montfort University, U.K., from July 2001 to July 2004. Apart from the degree that this thesis is currently applying for, no other academic degree or award was applied by me based on this work.

Glossary

Web-based System. A software system that resides on or interacts with a Web site on either the Internet or a corporate intranet.

Software Evoluton. A gradual process in which a software system changes into a different, usually more complex but better form. The research of software evolution is about building software systems capable of accommodating changes.

Legacy System. A software system that has existed for such a long time that a critical amount of business value has been accumulated, however, at the same time its implementing technologies have become outdated.

Software Reengineering. The process of applying reverse engineering, restructuring and forward engineering on an existing software system to extend its life time or functionality.

Abstraction. The process of eliminating or hiding known information to help understand a software system.

Reverse Engineering. The process to identify a system's components and their interrelationships and to represent the system at a higher level of abstraction.

Forward Engineering. The opposite process of reverse engineering, which gradually moves from a high level of abstraction to lower levels of abstraction.

Design Pattern. A recurring solution to a software design problem.

Framework. A pattern for an entire application which defines the structure of the application, its behaviors and control flows.

Software Architecture. A pattern for an entire software system composed of many applications, which defines the interactions between individual subsystems.

Software Architecture Recovery. A part of reverse engineering that aims at recovering the architectural elements of a software system.

Acronyms

ACS: Access Control Systems

ADL: Architecture Description Language

ADSR: Architecture Description for Software Reengineering

AMDA: Adaptable Model Driven Architecture

B2B: Business to Business

COOL: Common Object-Oriented Language

CSL: Common Structural Language

CSV: Comma Separated Value

DOM: Document Object Model

DTO: Data Transfer Object

EDI: Electronic Data Interchange

EJB: Enterprise Javabeans

IDE: Integrated Development Environment

ITL: Interval Temporal Logic

J2EE: Java 2 Platform, Enterprise Edition

JNDI: Java Naming and Directory Interface

MDA: Model Driven Architecture

MIS: Management Information Systems

MOF: Meta-Object Facility

MS: Manufacturing Systems

MVC: Model, View and Controller

ObTAM: Object-Oriented Temporal Agent Model

OCL: Object Contraint Language

PIL/PSL: Platform Independent Language/Platform Specific Language

PIM/PSM: Platform Independent Model/Platform Specific Model

RAT: Resource Allocation and Tracking Systems

RSS: Rich Site Summary, or Really Simple Syndication for RSS 2.0

RWSL: Reengineering Wide Spectrum Language

SAX: Simple API for XML

TDD/MDD: Test Driven Development/Model Driven Development

TGCL: Timed Guarded Command Language

WAE: Web Application Extension

WAF: Web Application Framework

WAI: Web Application Infrastructure

WSL: Wide Spectrum Language

XSLT: XSL Transformation Language

CHAPTER 1

Introduction

1.1 Software Evolution: Obstacles

The research on software evolution aims to develop software systems capable of accommodating changes over an extended operational lifetime [Warren99]. Apart from building systems from scratch, which is rarely the case, traditionally the dominant issue of software evolution is to incorporate new requirements into an existing system and/or make the system evolvable. This involves tackling the complexity of existing software systems that comes from their size and legacy techniques.

On the one hand, it is often too expensive or even impossible to completely replace a large software system developed a long time ago. Modern technologies directly supporting new requirements can not be easily integrated to them, thus software maintenance of these systems is limited to bug fixes or small functional enhancements that do not improve the overall system structure. On the other hand, it is increasingly difficult and expensive finding expertise in older technologies required for maintaining those legacy systems. As a result, the accumulation of small changes will turn out to be a great impact [Arnold92, Sneed88, Yang94 and Yang98].

For such legacy systems, software evolution, as an approach to transforming a legacy system to an evolvable one, is the only way to extend their operational lifetime and make them capable of accommodating changes. Such an evolution process relies on a sound understanding of the original system via not only the wisdom and experiences of software developers but various reverse engineering techniques.

Software reverse engineering has not made a significant advance over the last decade, compared to the booming development of software development technologies. The level of standardisation is the key reason that differentiates software reverse engineering from other software technologies.

Modern software development is built on a number of principles, paradigms, and tools. Those building blocks provide a standard, flexible and integrated way to develop and deliver a definite product. "Standard" means they are widely accepted by industry or/and academic. "Flexible" means alternative building blocks are always available to be used for a specific project. "Integrated" means different techniques should be combined to work together. "Definite" means the outcome of such development should always be a working software system in accordance with the requirements specification. Such a standard, flexible and integrated way for a definite product is called a unified solution in this research.

From programming language to operating system, from Integrated Development Environment (IDE) to software process model, no single product or theory is absolutely dominant. However, many alternatives can coexist and be regarded as "standard" due to their popularity or authority. To build a commercial web application, it is completely up to the development team to choose the operating system, programming language, IDE and development process. Thanks to standards built on techniques such as XML and UML, those building blocks could be integrated seamlessly and flexibly no matter how or by whom they were created. Finally, regardless of the technology adopted for development, the product of any software development should be always a working system, an instantiation of the requirement specification.

However, when it comes to software evolution, there is rather less consensus on those four adjectives. There is no standard, flexible and integrated way to evolve and deliver a definite product. Shall we use black-box or white-box reverse engineering for program understanding? Shall we produce data and control flow graphs, or some kind of formal specifications as the output of analysis? And it is often very difficult, if not impossible, to combine evolution tools from different sources to work together. Each of these techniques has its pros and cons of tackling various software evolution problems, but none of them on its own suffices to a whole evolution project. Apart from the limited interoperability between various tools, an often worse situation is that a specific tool can only tackle a specific programming language or platform.

A proper integration of various techniques capable of solving specific issues could be an effective way to unravel a complicated software system. This kind of integration has to be done from a standard point of view. The lack of standardisation is the biggest obstacle to software evolution.

1.2 Software Evolution: Web

When Tim Berners-Lee presented a proposal more than a decade ago for an information management system for sharing of knowledge and resources over a computer network, few people imagined how the so called World Wide Web would change our world.

The Web and Internet provide a superb platform for creating new business opportunities, which in turn keep inspiring a variety of innovations on both hardware and software systems supporting them. With the explosively exponential growth of its application, Web technologies have evolved from loosely connected sets of HTML pages to highly organised Web sites and to dynamic Web applications.

One implication of such prosperity is that there exist a great many similar, popular but not always compatible techniques. This diversity offers an even bigger challenge to the evolution of a Web-based system. Just imagine how many scripting languages available nowadays for server-side development. Being familiar with one of them could be good enough for developing a specific application, but certainly not for maintaining a large Web-based system with a number of applications built by different people in different languages.

In addition to the diversity, evolution of Web-based system is also hindered by the failure of the developers on applying software engineering principles proved in traditional software development. Such a failure can be attributed to all the aspects for building, using and maintaining a Web-based system, which are: user, developer, Web technologies and software engineering principles.

1.2.1 Issues about the Users

Web-based systems are highly distributed systems. Instead of having limited number of users operating on the system at a specific time, a large Web-based system could be

accessed concurrently be dozens or even hundreds of users from different geographical areas. As a result, software engineering phases like verification and testing are not followed in Web-based system development as strictly as in traditional software development due to the difficulty of simulating production environment.

1.2.2 Issues about the Developers

Not like traditional software development, one good thing about Web-based system development is that when a change needs to be made, chances are you do not need any recompiling or rebuilding for what has been modified. Usually replacing or modifying some HTML pages, JSPs, script files or other resources could suffice. Many such modifications look so straightforward and do not deserve to be carried out in a software engineering process.

1.2.3 Issues about Software Engineering

First, traditional object-oriented analysis and design method can not be fully applied on the development of Web-based systems. Many constituents of a Web-based system are not object-oriented, thus can not be modelled in that way. Second, the traditional software evolution method can not tackle Web-based systems in a unified way. Most of the previous research areas are focused on specific techniques which can not work together via standard interfaces to process a whole system.

1.2.4 Issues about Web Application Frameworks

While a poorly designed Web Application Framework is one of the biggest hurdles to building maintainable or evolvable Web applications, a good design needs to overcome challenges as follows:

- Compared to the changes of user requirements towards a Web-based system, changes made to the technologies used in building this system could present a bigger challenge to the evolution effort. Unfortunately, such changes, due to the various technologies required for building a large Web-based system, are often inevitable. It is the diversity and continuous changes of the software technologies that prevent an effective development and evolution of software systems. The advent of Web just exacerbates the situation.

- Compared to the building blocks of Web applications, their interfaces change much more frequently. For instance, the look and feel of a Web application could be customised for different user accounts, while all of them correspond to the same workflow. A successful Web Application Framework should allow an application to accommodate such change without the need to modify business logic or even web-tier control code.

- Compared to standalone applications, Web interfaces involve complex markup, the generation of which can easily obscure that of dynamic content. As two separate concerns of a Web application, markup and dynamic content should be independent of each other in a Web Application Framework to facilitate software maintenance and evolution.

- Compared to traditional UIs in languages such as Java, Web interfaces use a very different model that is restricted by requests and responses, where GUI components can not be dynamically updated as a result of underlying model changes.

The proposed Web Application Infrastructure and Framework addressing those issues are discussed in Chapter 5.

1.3 Research Method

Empirical approach centres on evidence derived from direct observation and experimentation in the acquisition of new knowledge [Kazdin03a]. This research is based on direct observation of existing software evolution methods, their application on Web-based systems, and experimentation in building new framework and tool for Web-based systems evolution.

It is generally agreed that the following elements are essential to a successful empirical approach: Observation, Experiment and Analyses [Marczyk05]. The research methodology adopted in this research will be explained in respect of those elements.

1.3.1 Observation

As an important element in any scientific research, observation implies two distinct concepts: being aware of the world around us and making careful measurements [Marczyk05]. The idea of this research and the ensuing questions came from the observation of the past achievement of software reengineering technologies and the present development of Web-based systems.

While legacy systems are the objects of observation for software reengineering, various Web application technologies are the objects of observation for Web-based systems development. The research of Web-based systems evolution, however, requires an observation of objects from both software reengineering and Web-based systems development. An important aspect of any scientific observation is measurement. Key concepts and terms in the context of their research studies need to be defined unambiguously via measurement.

The objects of observation and related measurement for key concepts in this research are defined as follows:

- Evolvable Web-based systems. An evolvable Web-based system will be capable of 1) accommodating changes to any of its components without having significant impact on others; 2) communicating with other heterogeneous systems with different data formats. The measurement for evolvable Web-based systems includes:

    - Web Application Infrastructure. The infrastructure must provide mechanisms for decoupling dependencies between applications and infrastructure, and externalising such dependencies from code into configuration files.

    - Web Application Framework. The framework must provide mechanisms for a clear and clean separation between components of a Web-based system. A clear separation requires specific constructs provided by the framework for various components, while a clean separation requires

every component independent of others in terms of compilation and run-time dependencies.

- Communication. Successful communication between heterogeneous systems relies on mutual understanding of formats and protocols used for the transferred information. Format and protocol can be represented in XML and various vocabularies.

- Reverse engineering techniques and Model Driven Architecture. Reverse engineering techniques will be reviewed and integrated into Model Driven Architecture to suit Web-based systems. The measurement for reverse engineering technique adapted for Web-based systems evolution in MDA environment includes:

  - Object Constraint Language. The abstraction rules should be defined in OCL syntax compatible with MDA model transformations.

  - UML. UML profiles should be defined to represent the results of reverse engineering as models compatible with MDA model transformation.

- Data mining techniques and Web-based systems. Data mining techniques will be reviewed and studied to find a way to understand Web-based systems. The measurement for data mining techniques for discovering relationships among components of a Web-based system includes:

  - Patterns taxonomy. Patterns taxonomy provides domain knowledge for calculating initial components, making adjustments during clustering or classification, and giving meaningful explanation to the results. The pattern taxonomy should cover the structure and implementation of Web-based systems as detailed as possible.

  - Relationship analysis. A system is a group of interacting and interdependent components forming a complex whole. One of the key issues in understanding a system is thus to accurately specifying the

relationships. The data mining technique developed in this research should be based on a relationship analysis approach suited for Web-based systems.

1.3.2 Experiment

As stated earlier, a standardised integration of various techniques capable of solving specific issues could be an effective way to unravel a complicated software system. This also applies to Web-based systems. Why has the standardisation of software evolution not happened yet? The answer might be simple: it will not happen, until there are standardised building blocks for it.

It is the purpose of this research to quest for suitable building blocks and experiment a blueprint for a unified solution to Web-based systems evolution. While those building blocks are chosen via observation and analysis, they are further tested in the following experiments:

- An XML transformation tool is built to act as an adaptor for heterogeneous systems with different input/out data formats.

- Web Application Infrastructure (WAI). An evolvable Web Application Infrastructure is built to support decoupling components and externalising component dependencies.

- Web Application Framework (WAF). An evolvable Web Application Framework is developed upon the proposed infrastructure to provide an improved Model-View-Controller (MVC) architecture.

- Case studies. A functioning Web application is transformed into an evolvable one based on the proposed infrastructure and framework.

1.3.3 Analyses

While the transformation tool, infrastructure and framework are implemented as software packages validated in experiments, other components of the proposed solution are analysed but their implementations are left for future research. The notations,

taxonomies and approaches presented in those discussions, however, are supported by published studies or previous research.

- Formal Method. A set of abstraction rules is designed to extract formal specifications from procedures/methods. The abstraction rules are an extension to those defined in RWSL [Yang98] and adapted to use OCL for logic syntax. At the current state of the art, some aspects of formal methods, such as inference rules and proofs of correctness [Wikipedia05], could demand significant resources to calculate, limiting the use of formal methods to specific components of a Web-based system, where the benefits of having such calculations makes them worth the resources.

- Data Mining. The weakness of formal method can be overcome by data mining techniques in analysing large amount of information often featured in Web-based systems. We are not only going to establish linkage between source code and application domain knowledge by applying abstraction rules level by level, but apply data mining techniques to map domain knowledge to source code. Once such mapping is found, linkage between application domain knowledge and source code will corroborated by the result of formal abstraction.

- Relationship and Pattern Analysis. The matching between domain knowledge base and source program can only be carried out with a thorough understanding of relationships between components in both code and domain levels. Software patterns existing in a software system come from two sources: system imposed patterns and user designed patterns. Both kinds of patterns can be classified into categories according to their abstraction levels and play an important role in discovering relationships between components at corresponding abstraction levels. Relationship analysis and pattern taxonomy at different abstraction levels are discussed and a set of principles is given.

1.4 Major Contributions

This research focuses on a unified solution to Web-based system evolution, which consists of a framework, a process and related techniques, followed by case studies used

for evaluation. Finally conclusions are drawn and further research directions are discussed.

The following issues are studied in this research:

- Understanding Web-based systems in a systematic way
- Keeping the consistency among different evolution phases
- Archiving interoperability with XML and MDA compliant tools [Compuware05 and IOSoftware05]
- Providing visibility for the results and
- Architecting evolvable Web-based systems

The major contributions of this research are as follows:

- A unified approach that addresses the evolution issue for not just a component of a Web-based system but for the system as a whole, not just a single phase but the whole life cycle of the Web-based Systems Evolution.
- A reference model that can be adapted to various real world systems via a number of design, system and architectural patterns classified in this thesis.
- An Adaptable MDA (Model Driven Architecture) that combines the power of generative Model Driven Development and Test Driven Development.
- Relationship analysis that can be applied on Web-based systems reengineering.
- An evolvable Web Application Infrastructure that eliminates container dependency (discussed in Chapter 5) from business logic of Web applications.
- An evolvable Web Application Framework that promotes a clean and thin Web tier based on the proposed infrastructure.

- An XML transformation tool that transforms between legacy files and XML documents.

1.5 Success Criteria

The aim of this research is to build a unified approach to issues encountered in the whole lifecycle of Web-based systems evolution. The following criteria are given to judge the success of the research:

- Can this approach handle the diversity of Web-based systems?
- Is the target system (either from development or reengineering) capable of accommodating changes?
- Can this approach be integrated with MDA and be applied to Web-based systems?
- Is the approach feasible for realisation? For example, is it possible to build a practical tool based on the approach?
- Is the approach capable for industrial-scaled systems?

1.6 Thesis Organisation

The rest of this thesis is organised as follows:

- Chapter 2 gives an overview of the background. It answers such questions as what is software evolution and what the evolution of Web-based systems requires.
- Chapter 3 gives an overview of the related work in the areas of software reengineering, architecture description language and Web Application Framework.
- Chapter 4 presents the infrastructure, framework and process of developing/evolving Web-based systems. The activities of this process are explained and related patterns are classified to fit different evolution phases.

The individual components introduced in Chapter 4 will be discussed respectively from Chapter 5 to 10.

- Chapter 5 presents the proposed Web Application Framework based on the infrastructure given in Chapter 4.

- Chapter 6 discusses abstraction, formal method and their use in software reverse engineering.

- Chapter 7 discusses relationship analysis that is the basis for using patterns and data mining techniques during software reverse engineering.

- Chapter 8 discusses patters that exist at various levels of Web-based systems.

- Chapter 9 discusses the application of XML in data management and transforming legacy data format.

- Chapter 10 presents the implementation of an XML transformation tool.

- Chapter 11 presents three case studies demonstrating the proposed approach to Web-based systems evolution. Data mining and patterns taxonomy are discussed in this research without being experimented in case studies. These components complete the whole picture of the proposed approach but are left for future studies. The case studies in this research focus on formal abstraction rules with OCL syntax, Web Application Infrastructure and Framework, and XML transformation tool.

- Chapter 12 concludes the thesis and discusses possible future research.

CHAPTER 2

Background

This chapter discusses the background of the Web-based systems evolution research. Section 2.1, 2.2 and 2.3 introduce the main concepts of this research: software evolution, patterns and Web-based systems. Section 2.4, 2.5 and 2.6 introduce the main techniques for modeling a Web-based system: XML, UML and MDA (Model Driven Architecture).

2.1 Software Evolution

A classic study by Lehman and Belady identified eight "laws" of system change [leh85]. The first two laws are defined as follows.

Continuing change — An E-type program that is used must be continually adapted else it becomes progressively less satisfactory.

Increasing complexity — As a program is evolved, its complexity increases unless work is done to maintain or reduce it.

Change and complexity are the main themes of software evolution. Software evolution is the process of gradual change of a software system during a whole lifetime, spanning from its birth, growth, operation until retirement. This definition implies the dynamic nature of a software system. It is dynamic during the analysis, design and development phases, for the system itself keeps growing and the user requirements are adjusted constantly. It is even dynamic after delivery, for the delivered software system inevitably will contain some latent defects that were not detected during testing or there come new requirements, in both cases the system needs to be modified during operation phase. Software evolution represents constant changes, which a software system must be able to accommodate. Compared to the pre-delivery evolution, the post-delivery evolution has much more impact on the operational lifetime of a software system. Software evolution is the main cause of software maintenance activities [Arthur98].

This research aims to solve two major issues of Web-based systems Evolution: reengineering a Web-based system and building an evolvable Web-based system.

2.1.1 Software Reengineering vs. Software Maintenance

Software systems will always have low-level maintenance requirements, and some of these changes can add value rather than merely maintaining the value of the software [warren99]. Software maintenance is characterised as: fine-grained, short-term and focused on localised changes. The implications of its characteristics are:

- The structure of the system remains relatively constant and the changes produce few economic and strategic benefits.
- There is a tendency to respond to one software requirement at a time.
- There are few economies of scale that accrue from software maintenance.
- There is little in the way of enhanced reuse.

Only through structural change can the software provide leverage for further development efforts. Thus we strive to increase the asset value of software by making it capable of accepting substantive structural changes.

Software reengineering is a coarser grained, higher level, structural form of change that makes the software systems qualitatively easier to maintain. Evolution allows the system to comply with broad new requirements and gain whole new capabilities. Instead of changing software only at the level of instructions in a higher level programming language, change is made at the architectural level. Software reengineering can increase the strategic and economic value of the software by making it easier to integrate with other software and making it more of an asset than a liability.

Reengineering improves software quality and makes the subsequent software maintenance tasks easier. Particularly, if reusable components are identified and the software is reengineered into a form that better supports reusability and maintainability,

the quality of reengineered software is considerably better than that of the original software.

The process of reengineering software systems involves three main steps: restructuring, reverse engineering and forward engineering [Warren99]:

- Program restructuring. Programs with a degraded structure are particularly difficult to evolve. Automatic program restructuring tools are now relatively common for popular languages such as COBOL and FORTRAN. These tools transform poorly structured programs to functionally equivalent programs coded in modern programming constructs. The resulting programs are more evolvable because they are easier to understand and change.

- Architecture restructuring. Program restructuring does not improve the system's architecture because individual programs are improved in isolation from the rest of the system. System level restructuring takes an architectural view of the software system and aims to make the structure clearer and more understandable. At this level, the relationships between components should be analysed. Improving a system's architecture is typically a manual exercise, because you need an understanding of the responsibility or function of each module. Where a legacy system comprises several programs with shared data structures, restructuring often propagates data reengineering due to the tight coupling between programs and data.

- Reverse engineering. Reverse engineering is the process of analysing software with the objective of recovering its design and specification. In most cases, source code is available to generate documentation using automation tools. Reverse engineering is typically the step prior to other reengineering activities. The results of revere engineering can provide engineers with a better understanding of a legacy system and can be used to determine whether any further reengineering is necessary.

- Forward engineering. The process of developing a system starting from the requirement specification and moving down towards implementation and deployment.

Software reengineering is closely related to software maintenance. Three reasons for software maintenance have been given in [Swanson76]:

- The errors in specification, design and implementation must be corrected (corrective maintenance).

- Second, the data and processing environments may change (adaptive maintenance).

- Third, the performance of the software must be maintained (perfective maintenance).

The ANSI definition for software maintenance provides the fourth reason besides these three ones. Software is changed to improve future maintainability or reliability or to provide a better basis for future enhancements (preventive maintenance) [Bendifallah87, Pressman94]. Preventive maintenance is closely related to re-engineering and each of thc other types of maintenance can be aided with reengineering techniques.

The documentation of software systems under maintenance is often incomplete or outdated. Reengineering can help recover such documents by producing structure charts, data flow diagrams, entity-relationship diagrams and system architecture views, all of which are essential for maintenance activities.

When to apply reengineering depends on the complexity and business value of the software system [Jacobson91].

If the legacy system has complete documentation or a relatively clear architecture so that it is easy to change, the common maintenance should suffice. If the system is difficult to change, but it has built up considerable business value, the system will need reengineering when significant improvements or new functionality are required.

Systems with high complexity and relatively low business value should be redeveloped or replaced.

2.1.2 Legacy Systems: Obsolete Techniques vs. Poor Development Practices

Traditionally, the complexities in maintaining a legacy system often result from its two main features:

- Obsolete Technologies. A large system with an over-stretched operational lifetime may end up with its underlying software infrastructure becoming obsolete, thus is not capable of meeting requirements of new software or hardware. And the expertise of people involved in the design of the original system may no longer be available.

- Poor Development Practices. As stated earlier, repeated modifications and updates to the original system are inevitable. The original architecture is more or less degraded. In addition, in many cases, those changes are not documented properly to keep the consistency between the implementation and documentation, because of lack of resources and time, or the reluctance of the developers to do so.

Most legacy systems have one or both of the features, the implication of which is that the system gradually becomes unmanageable. The business value accumulated in a legacy system keeps it from retirement, while any enhancement can not be easily accomplished and changes made in one part of a system may have a potential impact on the rest of it because of the lack of traceability.

2.1.3 Reverse Engineering: Formal vs. Cognitive

Techniques of reverse engineering can be classified into two groups. They are based on formal or cognitive methods.

2.1.3.1 Formal Method to Reverse Engineering

Formal methods aim to provide mathematical foundation for the design and verification of software systems. A formal method is typically composed of a specification language

with a mathematically established semantics and a development notion that defines how to verify a design (ultimately implementation) against its specification.

During reverse engineering, a formal method takes as input a source program and produces as output a formal specification. In the formal context, reverse engineering techniques can be classified into two categories: techniques built on knowledge-base or transformation library to produce formal specifications from code, and techniques built on derivation or translation to produce formal specifications from code [Gannod99].

A transformation transforms a specification to another form without changing its semantics. A program transformation applies on a group of programming statements. A translation does the same as a transformation but at an atomic level of granularity. They differ mainly on the degree to which high level knowledge about a problem domain or programming language is required during a transformation or translation process.

- Transformation rules transform aggregations of programming statements into simpler, equivalent sequences of statements or concise formal specifications. A large library of transformations is often required to capture various possible code constructions.

- Translation rules operate on single atomic statements such as assignments, conditionals, and iteratives to translate a program into an equivalent formal specification.

Both transformation [Ward89, Bowen91] and translation [Gannod95] have been addressed in formal method reverse engineering research. There has been limited industrial practice on this area though [Baxter97, Peritus and FermaT].

2.1.3.2 Cognitive Way to Reverse Engineering

Compared to formal methods, cognitive methods rely on pattern matching and supervised clustering techniques. A cognitive model describes the mental process or faculty of knowing a software system. A detailed survey in this area is given in [Mayrhauser95], where a comparison is made for six cognitive models of program understanding.

A hierarchy of cognitive design elements to support the construction of a mental model to aid program understanding in software-exploration tools has been defined in [Storey97]. One part of the hierarchy explains how to improve program understanding by supporting the actions of identifying software artifacts and the relations between them, by browsing code in delocalised plans, and by building abstractions. These actions comprise canonical reverse-engineering activities.

Two common approaches to program understanding are a functional approach emphasising cognition by what a system does and a behavioural approach emphasising how a system performs.

- The functional approach is bottom up and deductive, relying more on the knowledge of the implementation domain to produce higher level of abstractions that may map to the application domain and the system's functional requirements.

- The behavioural approach is top down and inductive, using hypothesis postulation and refinement to match artifacts derived from knowledge of the application domain onto the related software system.

Both top-down and bottom-up comprehension models have been used to guide software engineers to understand a software system. In reality, the two models are often used simultaneously for large system maintenance [Mayrhauser92]. This combined approach involves a repetitive process of making, testing and adjusting hypotheses until the entire system can be explained via a consistent set of hypotheses.

In this research, we employ a combination of two data mining techniques to achieve cognitive reserve engineering of heterogeneous information in a Web-based system.

Clustering and classification are two cognitive ways in data mining to understand a software system.

- Clustering analysis [KR90, Fis95, NH94] groups together data elements that hold similar attributes. The task of clustering is to learn a classification from the data.

- Classification analysis [MAR96, CS96, HCC93, and WK91] classifies new data elements to a particular group according to a profile that defines the common attributes of its members. The task of classification is to learn to assign instances to predefined classes.

The basic difference between these two concepts is that in clustering the data elements are unmarked. They do no belong to any group as far as the mathematical analysis is concerned. In classification, however, the data elements are marked. Clustering represents an unsupervised task, i.e., the training data doesn't specify the clusters, while classification represents supervised learning, i.e., the training data has to specify the classes.

Clustering is a classic area of machine learning and pattern recognition [Morgan72]. It groups instances that are close or similar to each other, and hence can be tackled in the same way. By visualising the data to provide an overview of the data attributes, clustering is well suited as the first step in building a model. However, a few complications arise in the domain of Web-based systems. The essential problem is that there is little consensus on which clustering algorithm to use for a given Web-based system. This is partly due to the variety and complexity of Web-systems. Different similarity measures have to be applied for different parts of the whole system, and it is difficult to define the relationships between different similarity measures without a thorough understanding in the Web-based system model. This issue is addressed in this research.

Once the taxonomy is created by clustering dataset, it is necessary to maintain it with example attributes for each class as the system changes and grows. This kind of maintenance may be greatly assisted by supervised learning, or classification, which trains a classifier with a corpus of files that are labelled with classes. At this stage, the classifier analyses correlations between the labels and other file attributes to form models. When the classifier is given unlabeled instances it can predict their classes reliably.

2.2 Patterns in Software Evolution

2.2.1 Pattern Form

To use patterns, for either development or reengineering, an appropriate representation has to be established. The presentation forms of patterns have evolved and branched in several directions. There are a variety of pattern forms available. A few of them are shown as follows:

- **Alexandrian Form**. This form is fairly close to the one used in [Alexander79]. It contains the sections Title, Problem, Discussion, Solution, a Diagram, as well as prologues and epilogues that connect this pattern to other relevant patterns.

- **Canonical Form**. Canonical form does not necessarily mean the original form, it means the simplest, most basic or primordial form. It is more formal and complete than the Alexandrian form, containing additional sections on Context, Forces, Resulting Context, Rationale and Known Uses. It is also known as "Coplien Form" as Jim Coplien was one of the more prominent pattern-writers to use it early.

- **GoF Form**. This form was used in the classic book by Gamma et al. [Gamma95], and uses different headings from the Canonical Form. However, most of its sections can be mapped to the canonical form.

- **Compact Form**. While many other forms try to structure and present as much information as possible, the Compact Form goes the other way and targets patterns that can be expressed on a single page. It contains the bare minimum: Context, Problem, Forces, Solution and Resulting Context.

- **Cockburn PM Form**. Alistair Cockburn used this form for his project management patterns. It is more verbal and oriented towards people and processes rather than technical software problems. It has many headings: Title, Thumbnail, Indications, Contraindications, Forces, Do this, Side Effects, Overdose Effect, Related Patterns, Principles, Examples and Reading.

- **Beck Form**. Kent Beck made his own variant of the pattern forms, using the headings Title, Context, Problem, Forces, Solution and Resulting Context.

2.2.2 Pattern Language

A pattern language is a collection of patterns that solve the prevalent problems in a particular domain and context, and, as a language of patterns, it specifically focuses on the pattern relationships in this domain and context. As an element of language, a pattern is an instruction, which can be used, over and over again, to resolve the given system of forces, wherever the context makes it relevant [Alexander79].

As with the instantiation of individual patterns into actual code or behaviours, the construction of statements in a pattern language is up to the developers. They manipulate patterns of a pattern language like classes of an Object-orient language. Various patterns are weaved together in different ways, at any given point to solve a specific problem. Such a pattern language is more than a collection of patterns.

2.2.3 Pattern Complexity

Patterns are applied in software development at different levels of complexity, three of which are given as follows:

- Pattern is a solution to a relatively small software problem that developers face repeatedly in the construction of an application. Patterns are used to solve modeling problems during analysis phase and construction issues during design phase.

- Framework is a pattern for an entire application. A framework defines the structure of an application, its specific classes and corresponding behaviours, and its control flow. Besides it provides extension points, through which custom classes can be added to the framework for application specific requirements.

- Architectural pattern is for an entire system composed of many applications. An architectural framework could be a group of application frameworks, but it does not have to be. Architectural patterns define how the individual subsystems work

together by providing guidelines to specify responsibilities and interactions for each of the subsystems/applications.

2.3 Evolution of Web Development

The design of Web interfaces plays an important role in building maintainable and evolvable Web applications. While there are numerous techniques to choose in Web development, those based on Java dominate the area of Enterprise Web-based systems and are typically helpful to understand the evolution of Web development and thus the problems to be avoided when designing new applications or even new frameworks.

2.3.1 Servlet-based Architecture

As the first Java web technology, Servlet API was released in 1997 to replace the then dominant technology, Common Gateway Interface (CGI) standard for server-side development.

Servlets enable Web development in a true OO way and perform well at invoking business objects. However, complex markups are still embedded in Servlets like those in C or Perl code for CGI, so that changes to the presentation of web content always demand modification of Java code. The old problems for CGI not addressed by Servlets are as follows:

- Any change to presentation will incur a Java code recompilation to make the change take effect.
- Java developers need to be involved at all times during the lifecycle of a Servlet -based application.
- Mixing markup and Java objects makes Web applications difficult to understand and maintain.

Additional infrastructures were required to address the above issues of Servlet-based architecture to separate markup from Java code.

2.3.2 JSP Model 1 Architecture

As a standard templating solution, JSP is endorsed by Sun to provide a markup language for dynamic Web application development. Unlike Servlet-based architecture, JSP pages can embed Java code and apply changes automatically with the recompilation done by the Web container.

The problem with JSP derives from its ability to allow embedded Java code (scriptlet). A J2EE web application's interface implemented only by JSP pages will map incoming requests to specific JSP pages. Request parameters are used to build domain objects that delegate tasks to business objects. Each JSP is also responsible for rendering the result of the business processing.

The above architecture for Web applications is classified as the JSP Model 1 architecture. On the one hand, this architecture is relatively easy to implement. On the other hand it is also easy to result in poorly designed applications with two main problems as follows:

- JSP handles application workflow by embedding complex or illusive dispatching logic into JSP pages, making it difficult to follow the executing path. In addition, mixing dispatching logic and result rendering in JSP pages is confusing and error prone.

- While embedded Java code (scriptlet) is hard to test and not reusable, mixing it with markup makes applications hard to maintain for both Java and markup developers.

Therefore, JSP Model 1 architecture results in non-maintainable applications and should never be used for large applications unless for prototyping.

2.3.3 JSP Model 2 Architecture

Both Servlet-based and JSP-based Model 1 architectures lack necessary mechanisms for separating responsibilities between Java and markup developers, which is of significant importance to large Web-based applications. The two technologies, however, can be

combined to address this issue, i.e., Servlets to handle control flow, and JSP pages to render page content.

This architecture is referred to as JSP Model 2 architecture that specifies a controller servlet as the central entry point, rather than individual JSP pages. The controller is responsible for accepting requests and directing application workflow, e.g., choosing a JSP to render content according to request parameters and the result of the business process handling. The view technology of this architecture is not limited to JSP, but could be any templating technology, while the controller is implemented as a Java Servlet in J2EE environment. With alternative view technologies, such as Velocity [Velocity], this architecture is referred to as Front Controller pattern (8.2.1.3) and is the key to applying the MVC architectural pattern (Chapter 5) to Web Application Framework for building maintainable and evolvable Web applications.

2.4 XML and UML in Software Evolution

XML and UML have been heavily used throughout modern software development. Their use has become almost de rigeur for software practitioners and is increasingly seen as mandatory for a genuine evolvable system.

An evolvable system is capable of accommodating changes over an extended operational lifetime. Such systems are designed explicitly for change and there is no end to evolvable system development, which is a significant departure from traditional develop-and-maintain model and advocates continuous improvement of software systems [warren99]. An evolvable system is a target of modern software development.

So what, exactly, is a legacy system? How to judge the evolvability of a system? A software system built dozens of years ago seems undoubtedly a legacy system, then what about an application developed a month ago using up-to-date integrated development environment? Is it definitely an evolvable system? Most legacy systems were developed using procedural languages such as C, COBOL, Pascal, while most modern systems are being written in object oriented languages such as Java, C++ or C#. Using a specific programming language itself, however, is not bound to make a legacy

system. For example, Web Services techniques allow a service to be built on either procedural or object oriented languages.

Using system duration or implementing languages as a metric is therefore insufficient for defining a legacy system unambiguously. For the purposes of this discussion, a legacy system is referred to as any working system that firstly does not provide native support for XML (that is, the system can not produce XML documents as output and consume XML documents as input), and secondly does not provide UML views at different levels. Under this definition, the process of transforming a legacy system to an evolvable one is thus a process of XML and UML enabling in a consistent way.

2.4.1 XML for Information Exchange

One of the applications of XML is for information exchange. The popularity of using XML as exchange format comes from its inherent advantages such as separating content and representation, implementing validation and stylesheets in XML too. The most important advantage is, however, that XML is the de facto standard for information exchange and has been adopted as the foundation for data exchange by almost all software products published recently. This XML role of exchanging information is a significant part of transforming a legacy system into an evolvable one.

For example, the Java 2 platform Enterprise Edition (J2EE) and B2B (Business to Business) are two popular platforms for evolving a legacy system that will meet the requirements of an evolvable system that we stated earlier. The B2B architecture supports widely distributed business objects loosely coupled with XML messages, while the J2EE architecture supports components that encapsulates business logic and resides in a container providing the runtime environment and other services such as transactional support. The core mechanism of the B2B architecture is an XML messaging system containing a set of business processes and the XML content describing the constituent transactions, a data dictionary description of the elements that make up the XML content, and a messaging service that specifies how the XML content is packaged, transferred, and routed. To make a distributed system that has its components loosely coupled and provide enterprise functionality, it makes sense to combine the two architectures by decomposing each business object using the J2EE

architecture and connecting business objects using the XML messaging system of the B2B architecture [Seacord03].

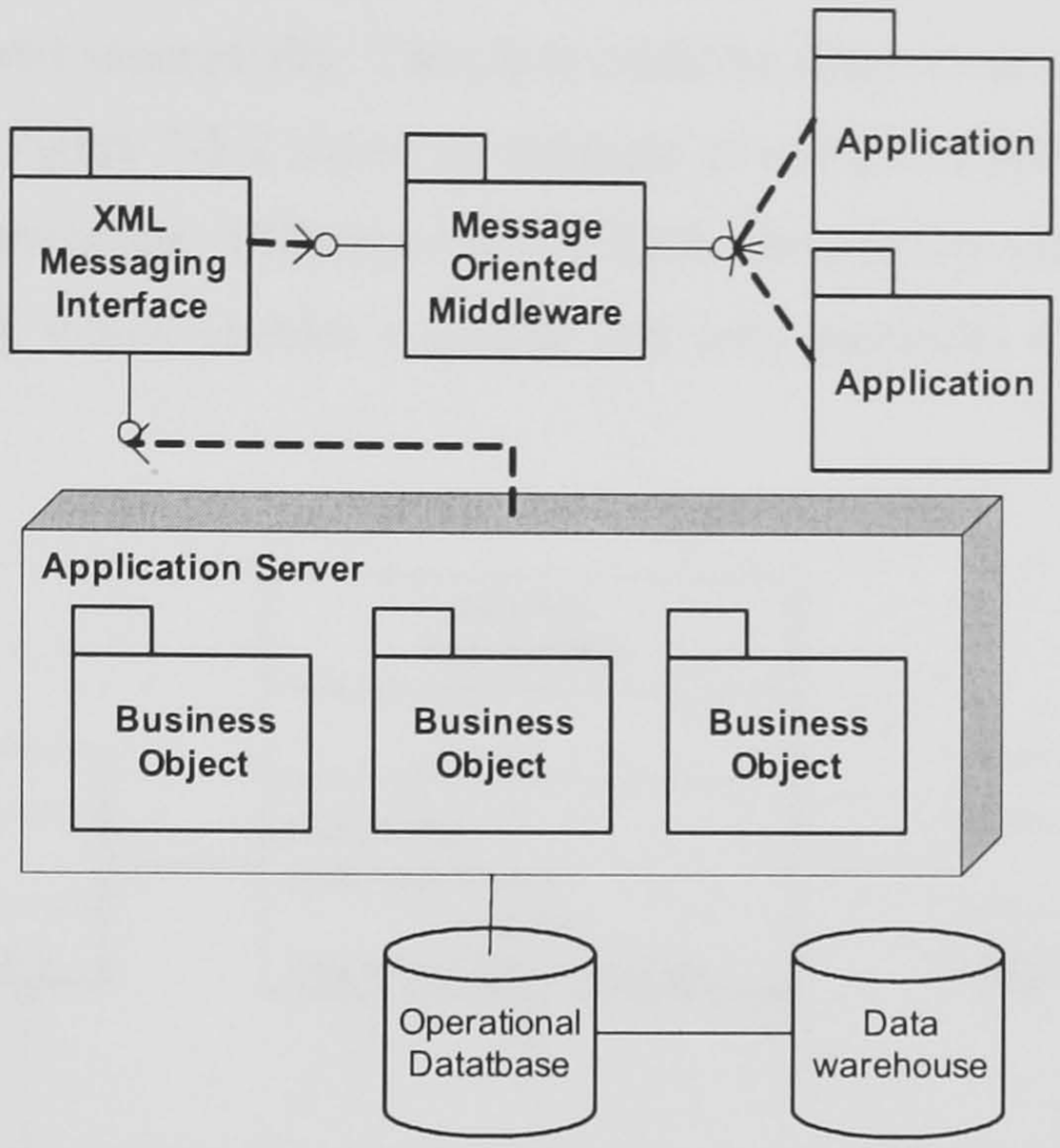

Figure 2-5-1 Evolvable Systems in B2B and J2EE

Figure 2-5-1 shows the relationship of business objects, where message-oriented middleware (MQSeries) is used to implement inherently loosely coupled, asynchronous communications. XML Messaging is used to implement business-to-business (B2B) and application-to-application (A2A) transactions and for connecting internal systems. J2EE defines a standard for developing multi-tier enterprise services that are highly available, secure, reliable, and scalable.

2.4.2 XML for Representation

Besides information exchange, XML can be used in a number of roles, for example, lightweight data storage, configuration script files and electronic forms. A complex system could contain more than dozens of file formats, some of which might be outdated and would better be replaced by new formats when transforming the system into a new architecture. One way to transform legacy files into XML documents is to use a Pipes and Filters architecture. 'Pipes and Filters' is an architectural pattern in which data in a standard format is passed through a series of components (filters) that

transform it in some way. The output of one filter is connected to the input of another via a connector (pipe). The filters are independent of each other. There are two interfaces (in and out) on each component and a pipe is mandatory between each pair of filters that are ordered sequentially. The pipes could be files, in-memory data structures or database tables. Figure 2-5-2 shows an example of using this pattern to convert X12 Electronic Data Interchange (EDI) standard to XML, then to Comma Separated Values (CSV) [Rawlins03], which enables a system that only processes CSV files to handle EDI documents.

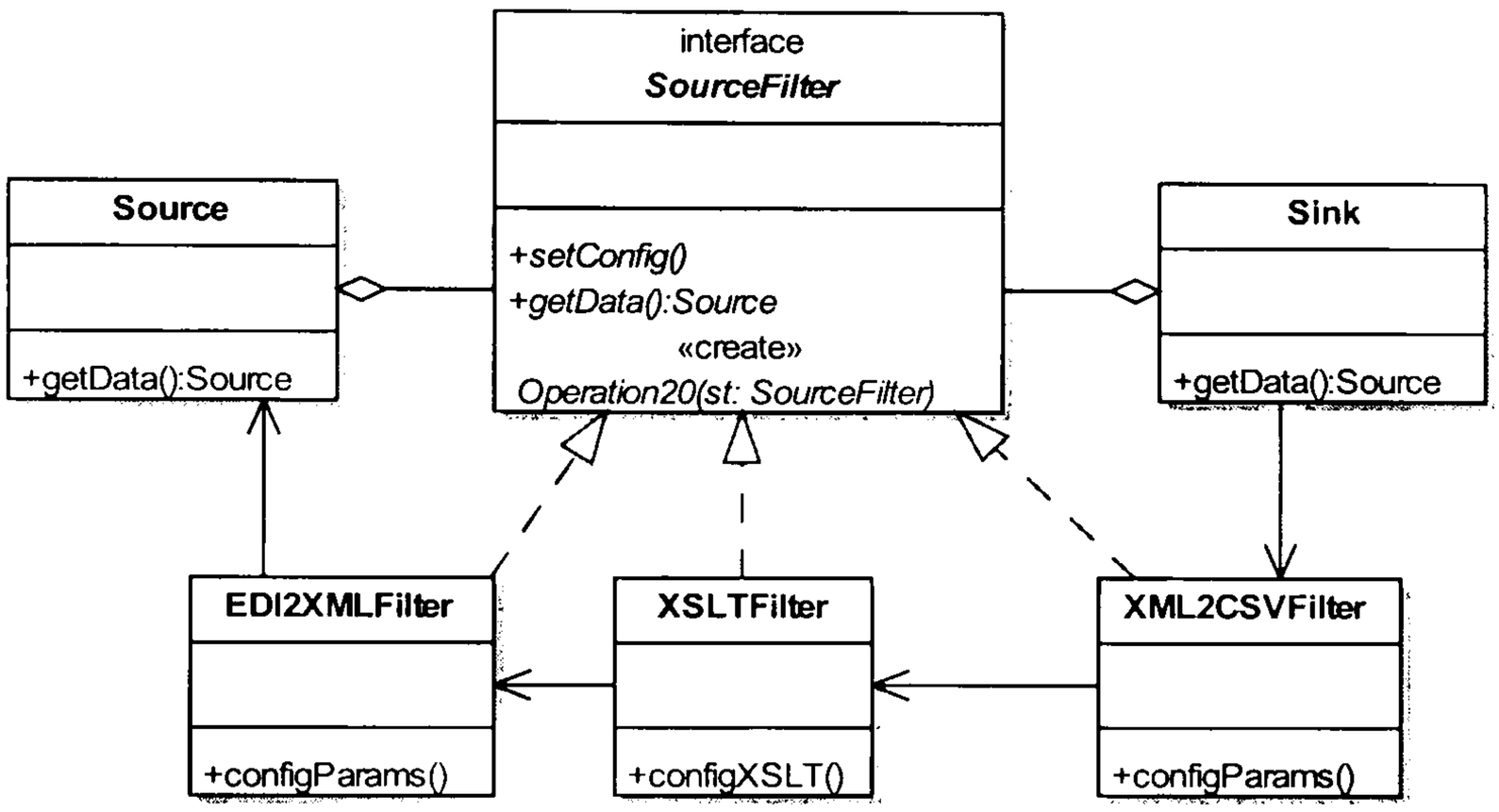

Figure 2-5-2 Pipe and Filter Pattern

2.4.3 UML for Representation

XML is, as we have stated, the de facto standard for information exchange and can be used to represent whatever you want. It is, however, designed for machine and not straightforward for human reading. On the other hand, UML is the de facto standard for representing design in a graphical way. UML has three important aspects, each of which is indicated by its name, i.e., language, model and unified. The three aspects are explained as follows:

- Language. The UML is a language for, either formally or informally, specifying, visualising, constructing, and documenting the artifacts of a system-intensive process, which emphasises on a systematic view of the steps for producing or maintaining a system. Specifying is to create a model to describe a system.

Visualising is to use diagrams made up of different notations to represent the model. Constructing is to transform a visual depiction of UML to an implementation of the system. Documenting is to use models and diagrams to record the requirements and the system throughout the system-intensive process.

- Model. A model is a miniature representation or a pattern of something. It shows a subject in an abstract way and provides a common understanding of knowledge of a system and its requirements.
- Unified. UML is a widely adopted standard throughout software industry. It is a common platform to unify various software techniques, tools and practices.

Although UML provides a set of notations, such as Class diagram, for representing object oriented (OO) concepts, its use is not limited to OO systems. UML profiles (library extensions) enables customisation of the canonical core of UML syntax to represent systems in specific domains. Figure 2-5-3 shows a representation of a client-server architecture in which the UML profile for Web Application Extensions has been used. Notice in the diagram that both form and client page elements are not Object-oriented.

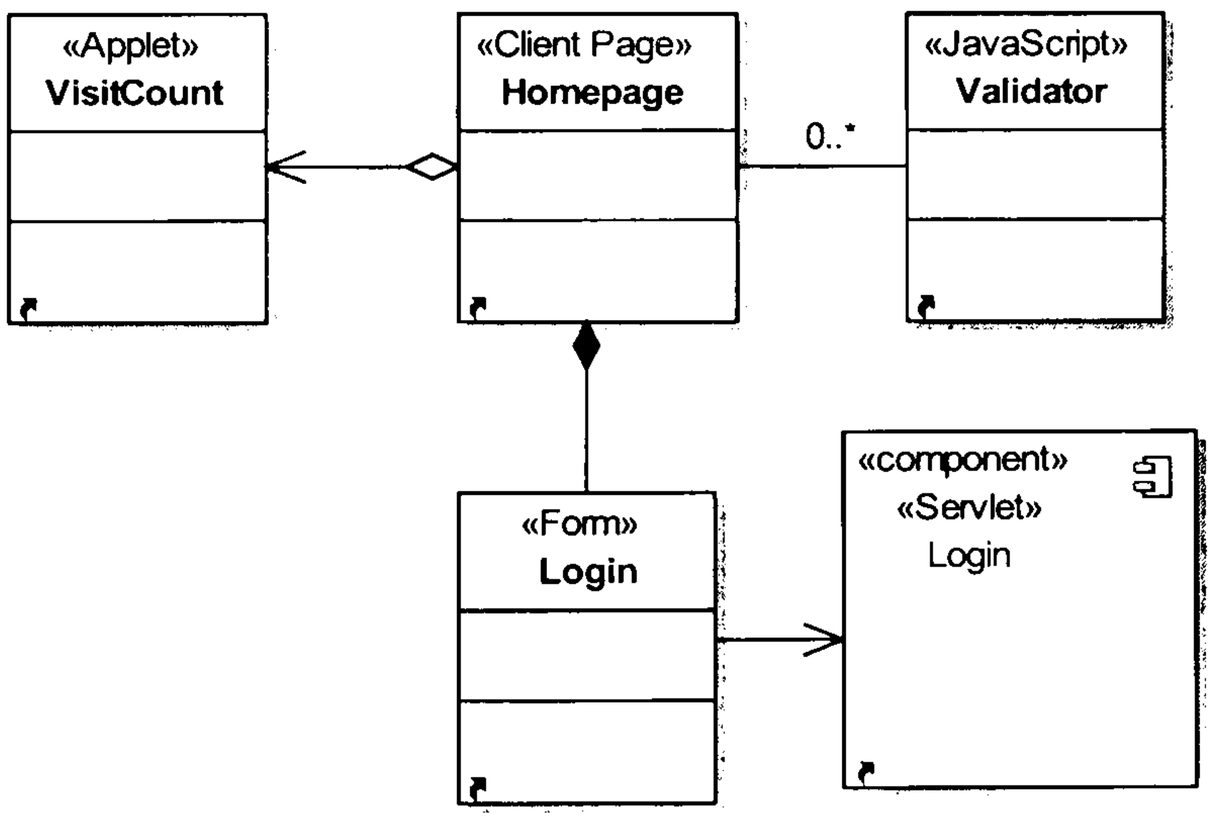

Figure 2-5-3 Client-server Architecture

2.4.4 UML for Automation

UML has built-in mechanisms for assisting automated software development.

2.4.4.1 Object Constraint Language

Object Constraint Language (OCL) is a formal Language [Warmer98], which is based on predicate calculus and allows for capturing constraints or rules and attaching them to model elements. Using a formal notation avoids the risk of misinterpretation due to an ambiguous natural language.

In previous UML versions, Object Constraint Language (OCL) was a standard way to express constraints with UML that could be interpreted by tools. There was the option of defining constraints using natural language or OCL. In UML 2.0 version, OCL is defined in its own metamodel. Although OCL is still not a necessity for every UML model, to use Model Driven Architecture or make the models executable, it provides the standard and sanctioned method for defining constraints on UML models.

OCL is a pure expression language. When evaluated it returns a pure value causing no side effects. The concept of well-formedness rules is an application of the OCL, which are built on the three constructs of OCL:

- The term precondition is borrowed from the concept "programming by contract". In a conventional contract, one party agrees to do something or provide something if and only if the other party fulfils his part of the contract. A precondition in OCL is a condition that must hold true at the moment that execution of a particular operation is about to begin.

- An invariant is an expression attached to a model element that must hold true for all instances of that model element.

- Post conditions also come from the concept of programming by contract. As the prefix implies however, the condition specifies something that must be true after the behaviour finishes. Post conditions provide the other half of the contract, the obligation of the invoked behaviour. A postcondition in UML is a condition that must hold true at the moment that execution of an operation has just ended.

OCL in UML can be used for the following purposes:

- Specifying initial attribute values
- Specifying the derivation rules for attributes or associations
- Specifying the body of query operations
- Specifying the targets for messages being sent
- Specifying guard conditions in statecharts
- Specifying end-user queries on a UML model

2.4.4.2 Action Semantics

Action Semantics, as defined in OMG document ptc/2002-01-09, is an extension of UML providing the metamodel for an action language. Action semantics describe imperative or dynamic behaviour, compared to OCL that is restricted to declarative assertions.

This addition to UML creates a standard way to describe dynamic actions completely, helping to ultimately generate complete implementations from UML models. In principle, using the action semantics one can write executable UML models shown in [Mellor02] and [Carter02].

Action semantics have been used mostly for real-time and embedded systems development. The advent of the UML Action Semantics standard will help make the action language approach more attractive in the area of enterprise system development. It is notable that the Action Semantics standard does not define how a standard concrete syntax for action statements should be implemented. Standardisation of the abstract syntax in the metamodel makes it possible for different tools to exchange models containing action statements even if they are built in different concrete syntaxes. However, the lack of standard concrete syntax may limit the practical development of this standard.

Without concrete syntax, there is no way to write any statement in the language in a standardised way. Several vendor-specific languages have been developed claiming to conform to the abstract syntax of the action semantics.

2.5 Model Driven Architecture

OMG's Model Driven Architecture intends to achieve language, vendor and middleware neutral software development. MDA focuses on various models existing in different phases of software development. A model could be a Computation Independent Model (CIM) or a Computational Model (CM). Both describe a system in a logic way. The former, also known as requirements model, does not involve technical factors, while the latter does. The computational model in MDA presents its most distinguishing features which exist in two forms: Platform Independent Model (PIM) or Platform Specific Model (PSM). PIM and PSM could be written in any formal language that make them understandable by both human and machines. PSM, however, is especially used by automatic tools and experts of a specific technology.

It is hard to describe the definite distinction between platform independent and platform specific without giving a specific context, i.e., the involved platform technologies. The terms PIM and PSM are relative terms. They only make sense when comparing models within such a context where a model is either more independent or more specific to the referenced platform technologies. Source code might be regarded as the most specific PSM. A series of PIM and PSM models construct a hierarchy of abstraction levels. MDA models at different abstraction levels can be transformed automatically to each other according to transformation rules (Transformation rules and mechanisms are 'first class citizens' of MDA modelling). Such transformation is carried out by automated tools that take a PIM as input and produce a specific PSM as output, and, theoretically at least, vice versa.

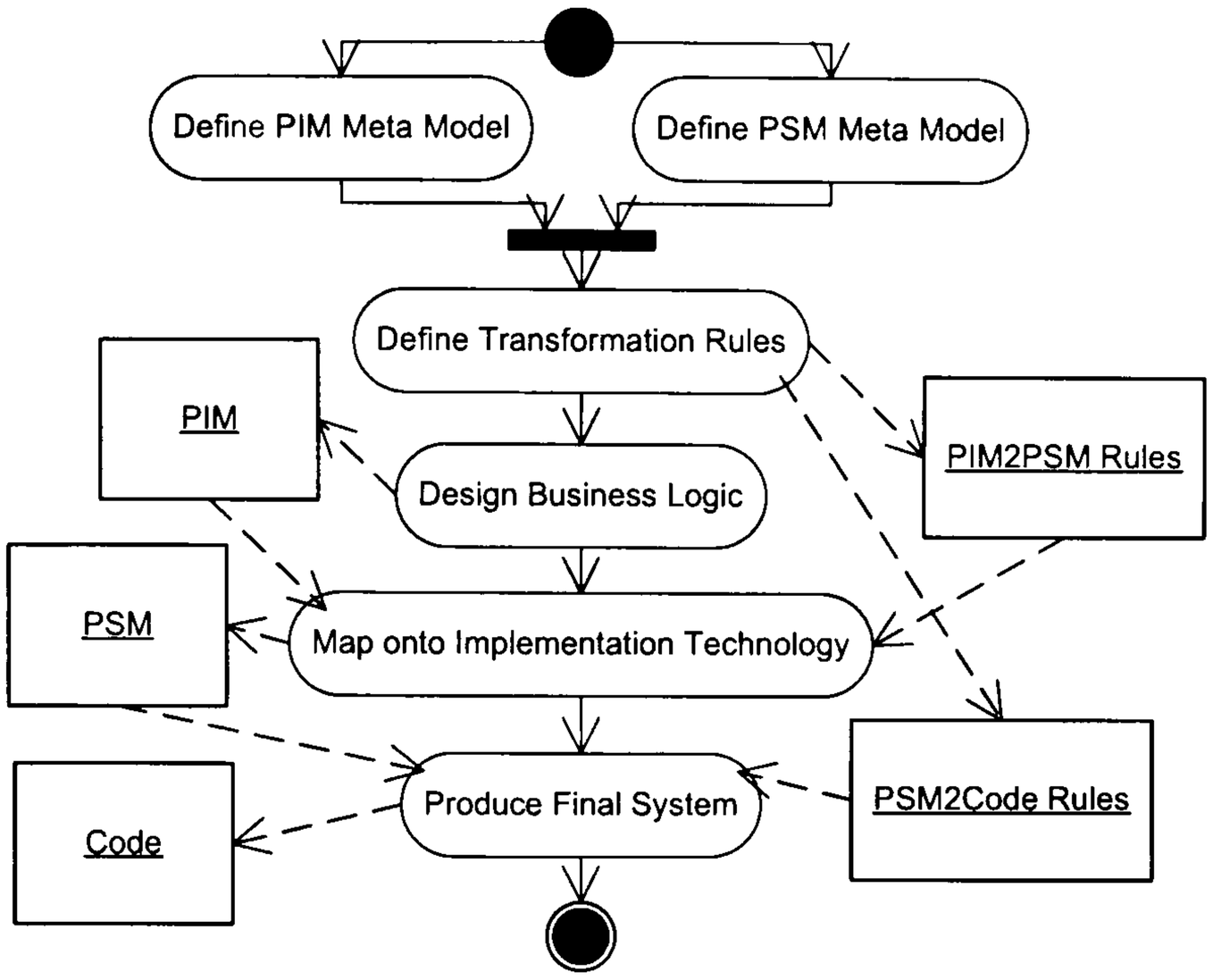

Figure 2-6-1 MDA process

Figure 2-6-1 shows the process of applying MDA. In MDA development, the first step is to create the platform-independent model (PIM), which is described in UML at different levels. The base PIM expresses only business functionality and behaviour, while the PIMs at the next level include some aspects of technology even though platform-specific details are absent. Secondly, the produced PIM is translated to a platform-specific model (PSM) using a UML Profile. This step will generate the application (e.g., J2EE, CORBA or .NET) from the PSM using a platform-specific code generator.

Although there is no restriction to the languages used to represent models of MDA provided they are conformant to the MOF (Meta-Object Facility) standard, UML is the favourite choice by far, especially for PIMs, for all the reasons described above [Frankel03].

2.6 Modeling Web-based Systems

For Web-based systems evolution, the increasing complexity has to be handled in an engineering way. One of the tasks in engineering a Web-based system is to model such a system. A number of methods have been researched [Lowe99]. In particular, many formalisms and models have been introduced to specify Web-based systems. Araneus

[Merialdo03], AutoWeb [Fraternali00], and WebML [Ceri00] present some interesting approaches in this area, where high-level models are used to design and develop each aspect of a Web-based system, ranging from requirements analysis, Web objects development to database design and presentation.

Those approaches use various often proprietary notation systems to model different aspects of a Web-based system. Some other approaches [Conallen03, Hennicker00 and Hennicker01] extend UML as the modeling language. The research of Web modeling is based on some fairly innocuous assumptions [Conallen99]:

- Web-based systems are software-intensive systems that are becoming more complex and more involved in mission-critical roles;

- One efficient way to manage complexity in software systems in development and maintenance is to abstract and model them. There is no exception for Web-based systems.

- A software system typically has multiple models, each of which represents a different viewpoint at different levels of abstraction. There is no exception for Web-based systems.

- The effectiveness of abstraction relies on the artifacts and developer activities in the development process; and

- UML is the de facto modeling language for software-intensive systems. This applies to Web-based systems as well.

2.6.1 Web Modeling Language

The Web Modeling Language (WebML) is a visual language for conceptual modeling of web applications. WebML also supports corresponding XML representation, which is used as a source for a generator to translate the model into a set of templates for the designed system.

WebML defines conceptual models of Web-based systems as follows [Ceri03]:

- Data model defines the data used by a Web-based system. It utilises the classical Entity-Relationship model with generalisation hierarchies, typed attributes and cardinality constraints.

- Hypertext model defines the organisation of the front-end interfaces of a Web-based system. It concentrates on a hyperspace topology of the system to define the organisation of the content using basic units, pages, links and unit composition within pages.

- Content management model defines operation units used to manage and update the content and invoke external programs.

In addition, the process of web-based application development with the WebML provides presentation specification for the layout and graphical appearance of Web pages.

2.6.2 Web Application Extension

A UML profile called Web Application Extensions (WAE) [Conallen99] is proposed for the design phase of developing Web-based systems constructed from Web server, network, HTTP and browser. These extensions are focused on the Web Page from the perspective of both server side and client side. Web pages are modeled using classes with stereotypes «server page» and «client page» respectively. The former represents a web page having server-side scripts which may interact with other components such as data broker, business logic, external systems, etc. A server page can be associated only with other server side objects. The «client page» stereotype represents a HTML page displayed in a browser; which may contain scripts to be executed on client-side. Client pages are associated with other client pages or with server pages. For the association between a client page and a server page, the «build» stereotyped association is used to indicate that a server page builds a client page. A Hyperlink between pages is modeled with «link» stereotyped association. The parameters, if any, of a hyperlink are modeled as link attributes of the corresponding association.

WAE defines a set of stereotypes, tagged values, and constraints for modeling Web-based systems. Appendix A shows stereotypes that may be applied to classes (on Class,

Component, and Use Case diagrams), components (on Component diagrams), and object lifelines (on Sequence diagrams).

2.7 Summary

The motive of this research stems from the background:

- The evolution of Web-based systems presents a bigger challenge to traditional software reengineering technologies.
- Clustering and Classification for data mining are suitable techniques for discovering information from Web-based systems.
- Patterns help facilitate not only software development but also software reengineering. This applies for Web-based systems too.
- UML and XML as de facto techniques should be integrated to any development areas that have the desire to communicate with others.
- Model Driven Architecture is the future trend for software development and should be integrated into software reengineering technology too.

The goal of this research is to address the above issues by developing the following techniques:

- Combination of Formal and Cognitive Methods,
- Combination of Clustering and Classification,
- Taxonomy of Web-based System Patterns,
- Standardisation of UML Profiles and XML Integration for Web-based Systems,
- Evolvable Web Application Framework, and
- Round-Trip Engineering (model to code and code to model) for MDA.

Those techniques will be discussed and examined in the remainder of this thesis.

CHAPTER 3

Related Research

This research aims to provide a unified solution to building evolvable Web-based systems, from scratch or existing systems. Two major tasks are implied in this statement: (1) understand an existing Web-based system and (2) implement an evolvable Web-based system, which are addressed in this research by developing new software reengineering techniques and Web Application Framework (WAFs). This chapter will review some existing software reengineering techniques and WAFs.

3.1 Software Reengineering

3.1.1 Formal Methods for Reverse Engineering

3.1.1.1 Maintainer's Assistant

The project, as the main part of a larger project ReForm funded by IBM and the DTI/SERC [Yang03, Yang91, Gerald20], aims to reengineer installed software so that modern software techniques can be adopted via the application of formal transformations.

Maintainer's Assistant (MA) employs transformation techniques to derive a specification from a section of code and to transform a section of code into a logically equivalent form. MA has features as follows:

- It acts, initially, on existing program code as a tool to aid comprehension (possibly by producing specifications) and only the program code is required for the processing;
- The system can work with any language by first translating, i.e., with a standalone translator into WSL (Wide Spectrum Language) and changes are made to the WSL program by means of transformation;

- The system incorporates a large, flexible catalogue of transformations. The applicability of each transformation is tested before it can be applied;

- The system is interactive and incorporates an X-Windows front end and pretty printer called the Browser;

- The system includes a database structure to store information about the program being transformed, such as the variables assigned to within a given piece of code;

- The system includes a facility to calculate metrics for the code being transformed.

One of the most important successes of Maintainer's Assistant is that it is based on a wide spectrum language, which defines syntax and semantics formally. However, Maintainer's Assistant focused on transformations rather than abstraction (semantic weakening). It involved very little in how to use multi-levelled abstractions and relevant abstraction rules to reach a good system reengineering, especially reverse engineering.

The Wide Spectrum Language in Maintainer's Assistant is sequential and non-distributed, which limits its application in domains such as Web-based systems.

3.1.1.2 PRISME

PRISME is a reverse engineering tool based on functional abstraction [Balmas96 and Balmas97]. The tool aims to re-document programs with outlines, which allow to contract the amount of information required to understand programs, facilitating developers to localise given computations or identify the role of a piece of code.

PRISME is a system for program re-documentation. It can automatically construct outlines of a subset of Lisp looping functions. The first step of program outlines approach is to build a model suitable for the representation of computations performed within loops. Such a model has both formal and conceptual features. However, the abstraction produced by PRISME is function-based instead of semantics-based. PRISME does not involve wide spectrum languages. It does not engage a mature formal method to specify the target system, therefore, PRISME can only extract simple

signatures as pieces of outline description of the system. No complete specification can be extracted in PRISME. Moreover, the notations in PRISME lack of integrated semantic foundation.

PRISME does not contain any abstraction rule to carry out its proposed abstraction for re-documentation and is only capable for a narrow subset LISP looping functions, not for applicable for the variety of Web-based systems, which are both procedural and object-oriented.

3.1.1.3 RWSL

RWSL (Reengineering Wide Spectrum Language) is a multi-layered wide spectrum language with sound formal semantics based on Interval Temporal Logic (ITL) [Moszkowski85 and Moszkowski86].

The architecture of RWSL is divided into two parts. One part is for object-oriented elements of a software system, which includes three layers, namely ITL Specification, Object-Oriented Temporal Agent Model (ObTAM) and Common Object-Oriented Language (COOL). ObTAM is an extension of Temporal Agent Model (TAM) language [Scholefield92] with objectoriented features. The most concrete layer of the object-oriented section is Common Object-Oriented Language, which provides structures as those in an ordinary OO language.

The other part is for structural (procedural) elements of a software system, which also includes three layers: ITL Specification, Timed Guarded Command Language (TGCL) and Common Structural Language (CSL). TGCL is an extension of Dijkstra's Guarded Command Language [Dijkstra76, Dijkstra90] with time and concurrency feature. Both TGCL and CSL are at the code level, while in CSL operators and concepts are implemented in common programming elements, such as shunts.

Both the object-oriented and procedural systems will be specified with ITL formulae. The semantics of other layers of RWSL, together with the abstraction and object extraction rules will be defined in ITL. The features of RWSL including the tool developed in the project -the Reengineering Assistant are as follows:

- use of ITL to define RWSL, allowing both non-real-time and real-time programs to be represented and manipulated;

- a small, traceable kernel language, i.e., ITL plus TGCL and ObTAM, allowing very precise and thorough formal semantics to be given to RWSL;

- transformation for all kinds of programs;

- object-extraction rules to enable transferring legacy procedural programs to object oriented programs;

- abstraction rules for crossing levels of abstraction in a stepwise manner and abstraction patterns as a means of describing current abstraction situations and acquiring expert observations of the target system, and then applying these observations in further abstraction;

- dealing with various languages via simple translation followed by automatic restructuring and simplification;

- an interactive, semi-automatic tool, rather than attempting complete automation, thereby making good use of human expert knowledge about the software and its domain;

- mechanical checking of the correctness conditions on transformation, object extraction and abstraction, appearing in the tool menus;

- using the prototype and manual case studies to demonstrate how the experienced user solves a problem, and then implementing these methods and heuristics and rapid prototyping development, with the system organised as a collection of abstract machines with formally defined interfaces.

RWSL is built on formal methods and supports both object oriented and structural elements of software systems. However it is mainly focused on real-time systems without sufficient semantic, graphical and architectural support for Web-based systems.

3.1.2 Software Architecture Recovery

Software architecture recovery is a kind of reverse engineering involving all activities for representing existing software architectures in an explicit way. Software architecture recovery aims to: recover lost architectural information, update architecture documentation, support maintenance activities, provide different views of architecture, migrate to other platform and facilitate impact analysis [Kri97].

3.1.2.1 Dali

The Dali architecture workbench [KC99] is an infrastructure for integrating a wide variety of extraction, manipulation, analysis, and presentation tools. It helps a user understand the architecture of an existing software system. Dali is an interactive system that aids the user in interpreting architectural information extracted automatically. The system does not attempt to automatically "find" the architecture for the user. Instead, Dali facilitates users to define architectural patterns and match those patterns to extracted information.

Three techniques of Dali are used in reconstructing the architecture of a software system:

- Architectural extraction capturing the architecture information from source artifacts such as code and makefiles. This static information can be combined with output from analysis tools that capture a system's dynamic behaviour such as profilers or test coverage tools.

- User defined architectural patterns, which map the system to its domain architecture consisting of abstractions used in architectural representations including subsystems, high level components, repositories, layers, conceptually-related functionality etc. Architecture patterns explicitly map these abstractions to the information extracted in step 1.

- Visualisation of recovered architecture as organised patterns.

3.1.2.2 Software Bookshelf Project

Software Bookshelf project [Finnigan97] provides for source model extraction a variety of extractors, such as C Fact Extractor (CFX) and C Information Abstraction (CIA). Manipulation and analysis of the source model stored in the repository is possible via tools like grep, sort, or Grok, to produce architecture information of a software system. Such information has a hierarchical structural based on system decomposition in terms of subsystems, files and functions. Architectures visualisation is supported via tools such as the Landscape Viewer.

3.1.2.3 Architecture Reconstruction Method (ARM)

Design patterns in software design help developers achieve high quality architectures. Reconstructing architectures of systems designed and developed with design patterns has traditionally been approached via manual source code inspections [Schull96].

A semi-automatic architecture recovery method, Software Architecture Reconstruction Method (ARM), is presented in [Guo99]. ARM consists of three major phases:

- Development of a concrete pattern recognition plan for extraction of a source model,
- Detection and evaluation of pattern instances, and
- Reconstruction and analysis of the architecture.

ARM reconstructs architectures based on recognised instances of design patterns. As an iterative and interpretive process, ARM incorporates human involvement as an integral part for evaluating the results and determining which patterns to apply in different iterations.

What differentiate ARM from other pattern recognition approaches come from its two features. First, ARM distinguishes abstract pattern description from concrete pattern instantiation which guides pattern detection. Second, automated tools used to perform pattern matching makes the pattern recognition process more robust compared to

manual inspections. During the reconstruction of the system's architecture, documented design patterns can be used to analyse conformance of the software system.

3.1.2.4 Software Architecture Reconstruction (SAR) Method

Software Architecture Reconstruction (SAR) method [Kri97] provides a formal method to software architecture recovery. It consists of such key elements as Relation Partition Algebra, architectural views, reconstruction levels, InfoPacks and ArchiSpects:

- Relation Partition Algebra. RPA is based on sets and binary relations. It has been defined to formalise descriptions of software architectures. Furthermore, in the context of reverse engineering, RPA offers abilities to express queries for structures in a formal notation, which can be executed on the model of a software system.

- Architectural Views (AV). Views are classified as: logical view, module view, code view, physical view and execution view. Scenarios [Kru95, KABC96] in the AV model support forward engineering as well as reverse engineering of software architectures.

- Reconstruction Levels. Each SAR level covers a range of architectural aspects that must be reconstructed. The following software architecture reconstruction levels exist: initial level, described level, redefined level, managed level and optimised level.

- InfoPacks and ArchiSpects. An InfoPack is a package of particular information extracted from the source code, design documents or any other information source. It contains a description of the extraction steps to be taken to retrieve certain software information. An ArchiSpect is a view on the system that makes explicit a certain architectural structure. It has a higher level of abstraction than an InfoPack. Most ArchiSpects build upon the results of InfoPacks. A complete set of ArchiSpects construct a system's actual architecture.

Each of these techniques on its own is insufficient to address the problem of architectural recovery for Web-based systems, where they do not have built in solutions and provide little room for extensions because of their lack of XML and UML support.

3.1.3 Web-based Systems Evolution

Web-based systems present new challenges on software reengineering techniques because of its complexity and usually gigantic size. To date there are only a few researches focused on reengineering Web-based systems.

An approach to reverse engineering Internet is developed in [Spring04] by annotating a map of the Internet with properties such as: client populations, features and workloads; network ownership, capacity, connectivity, geography and routing policies; patterns of loss, congestion, failure and growth; and so forth. Such combination of properties is greater than the sum of its parts, and exposes the attributes of network design easily overlooked by simpler, uncorrelated models. The feasibility of this approach relies on continuing improvements in measurement techniques, the potential to infer new properties from external measurements and an accounting of the resources required to complete the process.

In [Hassan03], various extractors are developed to analyse the source code and binaries of web applications. Algebraic manipulations are then applied on the extracted information. The output of this process is combined with input from domain experts to produce architecture diagrams. These diagrams highlight the key components of a web-based system and the interactions between them to enable a better understanding and easier maintenance. The discussed process for architecture recovery of web applications is semi-automated. The framework extends the Portable BookShelf (PBS) [Finnigan97 and Pen92] environment to address the differences between web applications and traditional software applications.

Other researches on reverse engineering of web applications include [Ricca03, Synytskyy03 and Tonella03]. Most of these efforts aim at reengineering the whole web site code implementing the application server and the user interface. This can be very

tedious even regarding the variety of techniques used in a commercial Web-based system.

Data mining has been successfully applied in analysing Web and Internet related information. However, very few researches have been done on how to reengineer a Web-based system with the combination of data mining techniques and traditional reverse engineering techniques. This is part of the research for implementing the proposed Web-based systems evolution solution.

3.2 Architecture Description Language

Architecture Description Language is defined in [Shaw95] with six properties:

- Composition and decomposition. The former is the process of integration of system components into larger sub-systems, while the latter the process of decomposition of a system into its constituents.

- Abstraction. A system has abstract views of high-level or low-level design.

- Reusability. Generic patterns of components and connectors are defined.

- Configurability. The structure of software systems can be changed independently from the components.

- Heterogeneity. Various architectural styles or programming languages can be accommodated in one system.

- Analysis. Architectural properties, metrics or simulating run-time characteristics are provided for analysing the system.

Research on Architecture Description Languages (ADL) can also contribute to the architecture recovery, since ADLs provide a clear goal for reverse engineering. In addition to the traditional, specialist ADLs, which are designed to use specific terminologies and notation systems, some researchers are building XML or UML based ADLs. In [Fuentes02], a UML profile is built for MultiTEL, a framework particularly well suited for the development of Internet-based multimedia and collaborative systems.

In [Dashofy02], an infrastructure is established for the rapid development of new Architecture Description Languages using an XML-based modular extension mechanism.

Before version 2.0, UML had limitations for implementing a complete ADL [Garlan00], while XML seems to be more powerful because of its flexibility. However, with the publishing of UML 2.0 specification, a UML-based ADL, which has a standard notational representation, formal semantics and the XMI (XML Metadata Interchange) format would be superior to a proprietary XML-based ADL because of its accessibility (through UML diagrams), accuracy (through Action Semantics and OCL) and portability (through XML/XMI).

3.3 Model Driven Framework

3.3.1 OptimalJ

OptimalJ is a MDA tool from Compuware [Compuware05] which supports the model driven development from the level of PIM. Its latest edition is Architecture edition 3.3 was tested in this research and it supports only J2EE platform but Compuware is also working on for support of .NET platform.

OptimalJ support the UML 1.3 version and it allows to model static and dynamic aspects of the system in different diagrams supported in UML 1.3 version. OptimalJ also allows the import and export of UML diagrams that are compliant to XMI 1.2 and UML 1.3.

OptimalJ supports 3 basic levels of models as Domain Model, Application Model and Code Model. Domain model corresponds to PIM and it further consists of two models: Class model and Service model. Class model covers the static structure of the system and application data whereas service model is used to describe behavioral aspects.

OptimalJ provides two levels of patterns for transformation, one pattern is Technology Pattern and second one is Implementation pattern. Technology patterns are used to define the transformation from Domain model to Application model (PIM to PSM), where as Implementation pattern corresponds to transformation between Application

model and code (PSM to Code). Patterns in OptimalJ are based on the proprietary Template Pattern Language (TPL). Using this language one write own patterns for model transformation and code transformation. OptimalJ provides very good support of predefined definition of transformation for J2EE. From a simple Class model, that represents PIM, one could have a running web-based J2EE application with basic functionality of add, update and delete of entities defined in the Class model.

3.3.2 Arcstyler

Arcstyler is a MDA tool from Interactive Objects [IOSoftware05] which supports the whole application life cycle of software development. Latest version of Arcstyler in the market is 5.0 and this version was also tested in this research.

Arcstyler support the UML version 1.4, OCL and model based designing and development of the systems. Arcstyler three ways of modeling the system requirements, one is creating models totally from scratch using UML, importing existing models that are compliant to XMI or creating models from existing java code for which term 'harvesting' is used in Arcstyler.

PIM in this tool is represented in terms of Class model where domain/business entities are represented as Class in platform independent manner. This class model is taken as a base for further transformation. Arcstyler does not provide any proper support for PSM where one could distinctly differentiate between PIM and PSM. Graphically it is quite difficult to sort out the PIM and PSM. Code model in Arcstyler is represented as subsystem where different source code packages are represented as components.

Arcstyler used the term MDA-Cartridges for the model transformation. Arcstyler has the support of creation and editing of MDA-Cartridges so that one could define or modify his own transformation rules. Arcstyler provides the predefined cartridges for a number of technologies and platforms, such as Java, J2EE and .NET. Using the cartridge concept one could define the transformation rules for a new technology or platform like CORBA, VB, VC etc.

3.4 Web Application Framework

Web Application Framework (WAF) is important to both software development and reengineering. Most MVC (Model-View-Controller, Chapter 8) web application frameworks comply with the following rules:

- A single generic controller servlet acts as an entry point for a whole application or part of an application, where request URLs are mapped in the standard web.xml deployment descriptor onto the controller servlet.

- The controller servlet delegates received requests to corresponding sub-controllers, depending on mappings often defined in an XML file as request URL/sub-controller name pairs.

- Sub-controllers choose views to display the results of business operations by specifying view names that are mapped onto specific view implementations.

3.4.1 Struts

As the most widely adopted MVC framework, Struts was originally written by Craig McClanahan, the main developer of the Tomcat servlet engine, and was released in mid 2000. Struts uses a single controller servlet for a whole web application or subset of a web application. Struts also comes with several JSP tag libraries to handle data binding and other operations. Despite its popularity, struts is far from an ideal J2EE web application framework for building maintainable and evolvable Web applications.

- In MVC WAF, controller should not contain business logic but delegate to business service layer to handle the actual business logic. In stead, the controller should focus on error handling and control flow. However, Struts does not provide infrastructure for enforcing the separation of business service layer and web-tier actions.

- In the Struts model, each Action object is given as a parameter an ActionForm object, which contains JavaBean properties pre-populated from the request using reflection. ActionForm objects are tied the Struts framework and thus cannot be used as commands in an application.

- Struts is too JSP-oriented, although it is possible to use Struts with other templating technologies.

- Struts is based almost entirely on concrete classes, which makes it hard to customise Struts' behaviour by pluggable implementations.

3.4.2 WebWork

Designed by Rickard Oberg, WebWork, first released in mid 2002, minimises the dependency of application code on web concepts.

While a Struts action is a reusable and thread safe object, a WebWork action is a command based on Command Design Pattern, created to handle a specific request. Like Struts, WebWork provides custom JSP tag libraries that are less closely tied to JSP than Struts. WebWork helps to minimise dependence of application code on the Servlet API and provides strong support for good design. However, it has disadvantages as follow.

- The creation of an action on every request could be verbose for requests with little data.

- Compared to using one reusable action, it is more difficult to configure many new actions or map them to business service layer.

- WebWork imposes the Command design pattern on every user interaction, where the types of exception a particular command may throw can not be predicted.

3.5 Summary

The current researches on Web-based evolution have covered a number of areas and made significant advances. However, most of them only apply to specific systems or phases in the process of Web-based system evolution. Even though some researches provide frameworks for the whole process, they do not provide underlying support for both XML integration and UML representation. There is not a solution to Web-based system evolution that combines formal and cognitive reverse engineering, Web modeling, XML integration and UML representation. In addition, existing Web

Application Frameworks do not meet the requirements for building maintainable and evolvable Web applications. For existing MDA tools, huge efforts have been taken on building strictly defined transformation rules and automatic transformations. However, the concept of pure Model based development is controversial and to some extent opposite to the increasingly popular paradigms of developing software, such as Extreme programming and Test Driven Development.

CHAPTER 4

A Proposed Approach to Web-based Systems Evolution

4.1 Software Evolution: Road to Evolvable Web-based Systems

This research is focused on establishing a general framework and methodology to facilitate the evolution of Web-based systems. Figure 4-1-1 shows an overall picture of the research methodology.

The left of Figure 4-1-1 shows the architecture of current Web-based systems which can be divided into such components as: server application, template application, configuration, communication and infrastructure/framework. There are some serious drawbacks in current Web-based systems shown above. First the existing framework does not enforce a clear separation of business and application layers. Second a Web-based system is often tightly coupled with the infrastructure that provides enterprise services. The configuration files within those systems are capable of connecting different components statically, but can not handle class instantiation and lookup. Finally the communication between different systems is hampered by their proprietary file formats delivered as messages, mails or commands.

The right of Figure 4-1-1 shows the architecture of Web-based systems built on proposed Web Application Framework (WAF) and supporting infrastructure, where the three components of server application, template application and infrastructure/framework are only connected via configurations. The configuration files of those systems not only establish relationships between different components, but provide information for instantiating classes to be used at runtime. The input/output of the proposed architecture are filtered by XML adaptors which conduct transformations between various file formats and XML documents.

In the middle of Figure 4-1-1 are the techniques used for transforming existing Web-based systems to the proposed WAF and supporting infrastructure. Formal abstraction,

data mining and XML transformation are three techniques to target different parts of a Web-based system.

- Formal abstraction is applied on application functions/methods with strong calculation/control logics to extract specifications of higher level abstraction.

- Data mining is effective in analysing heterogeneous information to produce a set of closely-related nodes.

- XML transformation is the centre of data communication with external systems.

The proposed approach to building evolvable Web-based systems will be elaborated in this and following chapters corresponding to the numbered elements shown in Figure 4-1-1:

1. Build Evolvable Architecture. The purpose of reengineering existing Web-based systems is to add extensibility and maintainability supported by the infrastructure and Web Application Framework. Chapter 5 introduces the proposed WAF and supporting infrastructure.

2. Extract High-level Specifications. Chapter 6 introduces formal method based reverse engineering of procedures/methods with strong calculation/control logics.

3. Build Heterogeneous Information Model. Chapter 7 introduces data mining based reverse engineering of heterogeneous information with various link types to construct a representation of a Web-based system.

4. Build Domain/System/Integration model. Reverse engineering techniques, either formal method or data mining, to some extent rely on successful pattern recognitions. Enforcing the use of design patterns in software development not only facilitates building professional software systems with less effort, but makes such systems self-documented, ready for reengineering. In reality, using design patterns in reengineering is not as effective as in software development. Legacy systems were rarely built with nowadays trial-and-error patterns and design patterns are often too fine-grained for constructing a domain model.

Chapter 8 introduces domain patterns, system patterns and integration patterns, which are used to identify models from existing Web-based systems and come from a study of existing software systems and other research results.

5. Build XML Data Management. Software systems are built on data and logics that manipulate data. An evolvable system needs an extensible and interchangeable data representation as well as an implementation of those logics in modern programming languages. XML is the de facto standard for data representation. Chapter 9 introduces XML data management in Software Evolution.

The rest of this Chapter will first introduce the concept of abstraction used to understand an existing software system and the reengineering process with MDA concept.

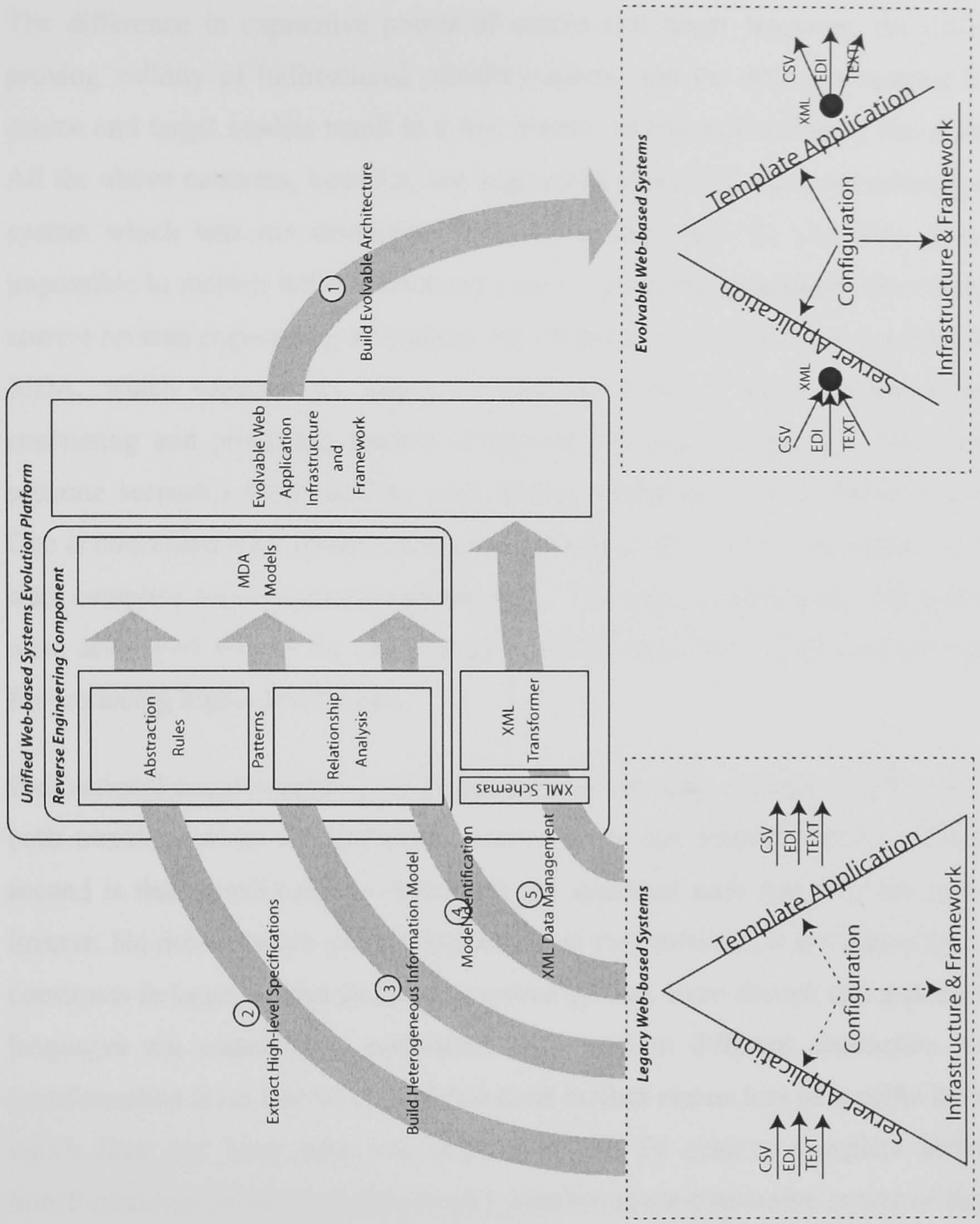

Figure 4-1-1 Methodology of Building Evolvable Web-based Systems

4.1.1 Round-Trip Engineering with MDA

Although MDA provides a framework for automatic transformations between models at different levels of abstractions, few researches have been done in respect of the reverse transformations between PIM, PSM and Code Models, which requires that "a transformation can be applied not only from source to target, but also back from target to source".

The difference in expressive power of source and target language, the difficulty in proving validity of bidirectional transformations, and the different abstract levels of source and target models result in a low priority of bidirectionality of transformations. All the above concerns, however, are aggregated especially when considering a legacy system which was not developed in MDA environment. In this case, it is almost impossible to recover MDA models by a set of predefined transformation rules, for the current reverse engineering techniques are either too limited or not compatible with the MDA, which requires the ability of tackling different levels of abstractions and consuming and producing models written in formalised languages. There exist two separate scenarios when dealing with reverse transformations in MDA environment. One is concerned with systems that were developed using MDA techniques and have at least complete forward transformation rules. The other is concerned with systems that were developed without the concepts of models in mind and require reverse techniques for producing higher level views.

Bidirectional transformations can be achieved in two ways [Kleppe03]. The first is that both transformations are performed according to one transformation definition. The second is that transformation definitions are specified such that they are each other's inverse. No matter which method is taken, it is very difficult, if not impossible, to map constructs in target system to those in source system. Even though two models in those languages are semantically equivalent, they exist at different abstraction level and transformation from low level to higher level models means loss of specific information which may not have their counterparts at all. To achieve complete bidirectional transformations between models is only possible if the expressive power of the source and target modeling language is identical, which means that the abstraction levels of both source and target model are equivalent. Since one of the essential characteristics for MDA is that PIM is at a higher level than PSM, making both kinds of models equivalent in expressive power is unacceptable.

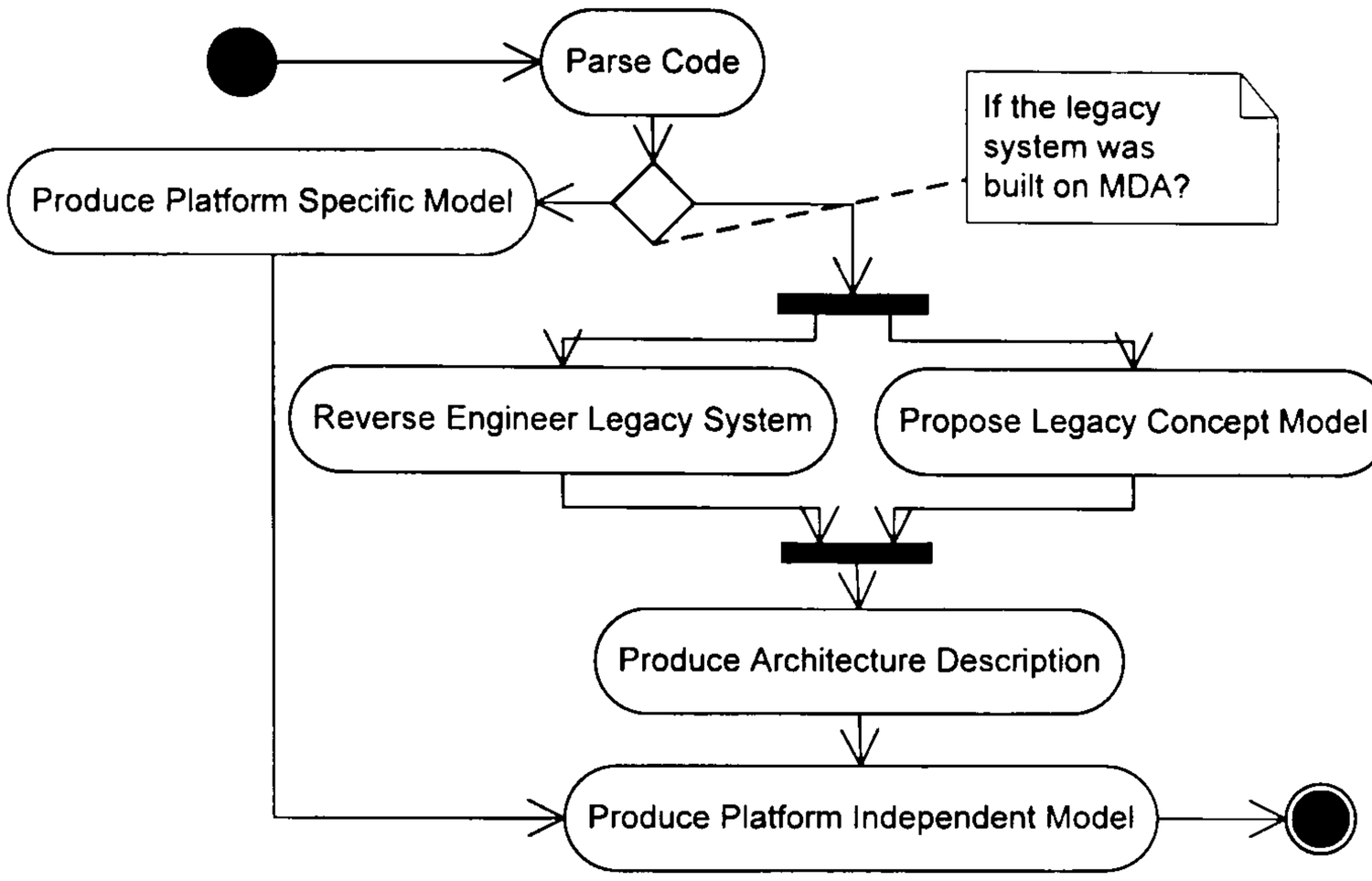

Figure 4-1-2. Reengineering within MDA Environment

It is therefore not easy to achieve bidirectional transformation even for a system built in MDA environment, and it's much more difficult to do so for a non-MDA system. However, there is often no need to transform a MDA compliant system reversely, for it conforms to the requirements of an evolvable system and could be understandable and maintainable as long as all the PIM documents available. The application of bidirectional transformations is actually in situations where the system was not developed in MDA environment and is now required to be transformed into an evolvable system that conforms to the standards of MDA. To achieve this goal, reverse engineering techniques have to be used to understand the legacy system and produce high level system views which could be implemented in any appropriate languages but should be eventually transformed into formalised UML diagrams to build PIM of the legacy system. Figure 4-1-2 shows the process of software reengineering within MDA environment.

Although there are a number of formal languages that could be used to represent models in MDA, it is reasonable to use the combination of UML and OCL to build such models, for there is a strong relationship between all OMG standards. In addition, OCL can be used to define the rules transforming those models.

4.1.2 Test Driven Development

Test Driven Development (TDD) is one of the core practices of Extreme Programming that aims to make the development process simpler, with short, continuous development cycles allowing constant feedback as to the state of the software. As MDA, TDD is not only applicable to software development, but also to software reengineering. It represents a simple evolution in the way software can be developed, where complex software systems can be developed in small simple increments that use tests to drive the design and implementation of the software.

Instead of creating a design to specify how to structure code, a test is created to define how a small part of the system should function. This test drives the design of the code needed to implement the functionality and allows a developer to discover the design as the code to make better decisions about the way the code is structured during development rather than up front.

TDD relies on two main concepts: unit tests and refactoring and consists of the following steps:

- Create a test that specifies how a small part of the software should behave.
- Run the test as easily and quickly as possible without being concerned with the design of code.
- Refactor the code if it is working correctly to remove any duplication or any other problems introduced to run the test.

As an iterative process, the above steps can be repeated a number of times until the new code is satisfactory.

4.1.3 Adaptable MDA

MDA aims for generative Model Driven Development (MDD), where sophisticated modeling tools are used to create sophisticated models that can be automatically transformed with those tools to adapt to various deployment platforms.

Pure generative MDD is, however, very difficult to achieve in near future for a complete automatic transformation of models demand standard, formalised model language, very comprehensive and sophisticated tool support. The situation is even worse for applying MDA on existing or legacy systems as discussed earlier.

The existing generative MDD, MDA, can not solve the real problems by its own. To apply MDA on a round-trip engineering, it needs to be more adaptable. No matter to what extent a software engineering/reengineering could be automated, human intervention is inevitable, where another principle other than strictly formalised rules needs to be applied. The weakness of MDA is discussed as follows:

- There is not a standard modeling language that satisfies real-world needs. There is a standard modelling language UML. However, even UML2.0 capable of extending diagrams via stereotypes does not address many fundamental issues ranging from user interface on the front end to a database on the back end.

- Successful application of MDA demands sophisticated tool support. A toolset together via the XML Metadata Interchange standard is supposed to be able to communicate with other tools. This requires tool vendors to strictly comply with the XMI specification, which is, according to past experiences with CORBA ORB interoperability, very unlikely.

MDA needs to be adaptable. An adaptable MDA (AMDA) needs to be applicable with both simple tools as well as sophisticated modelling toolset. Simple tools are inclusive, flexible, and not constraining. There factors are even more important for handling existing or legacy systems.

When it comes to detailed design modeling, sophisticated toolsets for object and data modelling can be used to generate code. AMDA advocates an evolutionary approach, where the simplest way to work should be adopted according to the target system and the goal and the code is written or reengineered iteratively and incrementally in step with the models.

MDA and TDD can complement each other in many aspects shown in Table 4-1-1.

	TDD	MDA
Feedback Loop	Shortens programming feedback loop	Shortens modelling feedback loop
Software Specification	Provides test-based specification	Provides model-based specification
Development Focus	Promotes high-quality code for participating developers	Promotes high-quality model for shareholders and other developers
Development Goal	Creates callable and testable operations	Solves design/architectural issues
Visibility	Not visually presented	Visually presented

Table 4-1-1 AMDA and TDD comparison

MDA and TDD should be combined together to create an Adaptable MDA to gain the advantages of both, where MDA should be used to create models with project stakeholders to help analyse requirements in architectural and design models and TDD should be used as a critical part of development efforts to ensure clean and working code. An AMDA will facilitate a high-quality, working engineering/reengineering process that meets the actual needs of new or existing systems.

4.2 Software Evolution Process

This section presents the evolution process, which is defined independent of platform specific technologies and needs to be instantiated when applied on a software system.

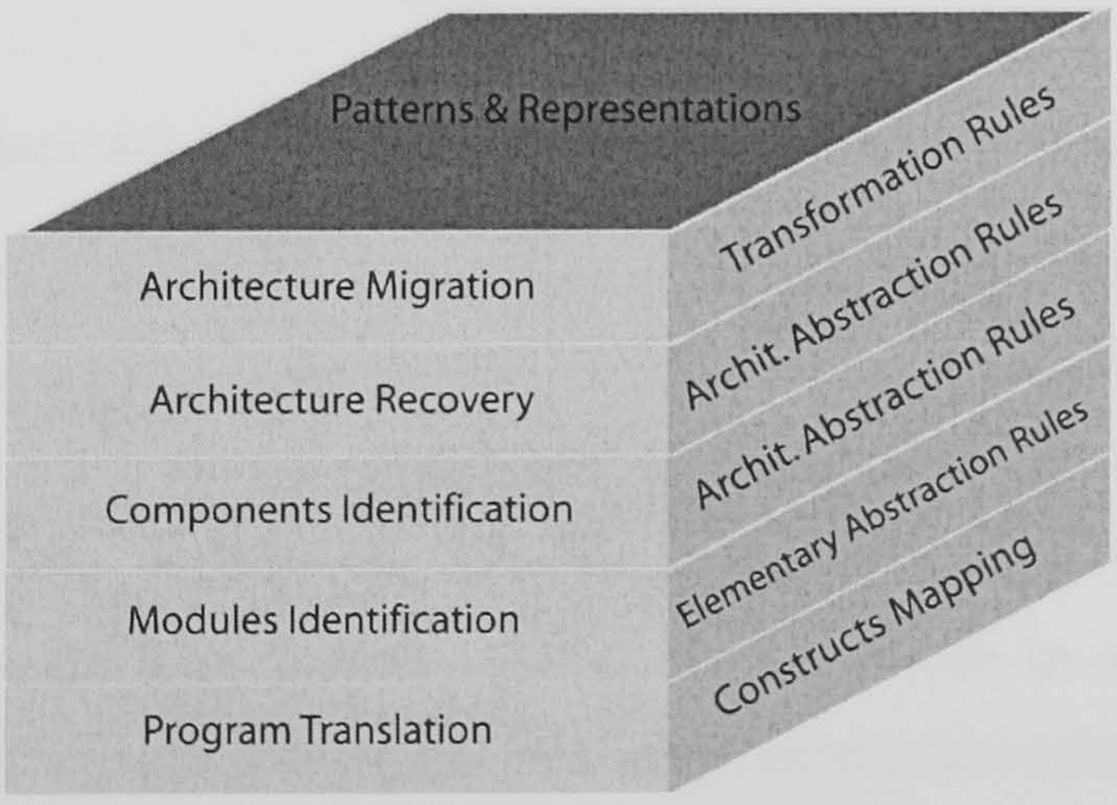

Figure 4-2-1 Cubic Model for Software Reengineering

Figure 4-2-1 shows a cubic model, which consists of five levels: program translation, modules identification, components identification, architecture recovery and

architecture migration, with their corresponding patterns and representations used during processing. The lower four levels actually stand for four abstraction processes, the outcomes of which present different system views.

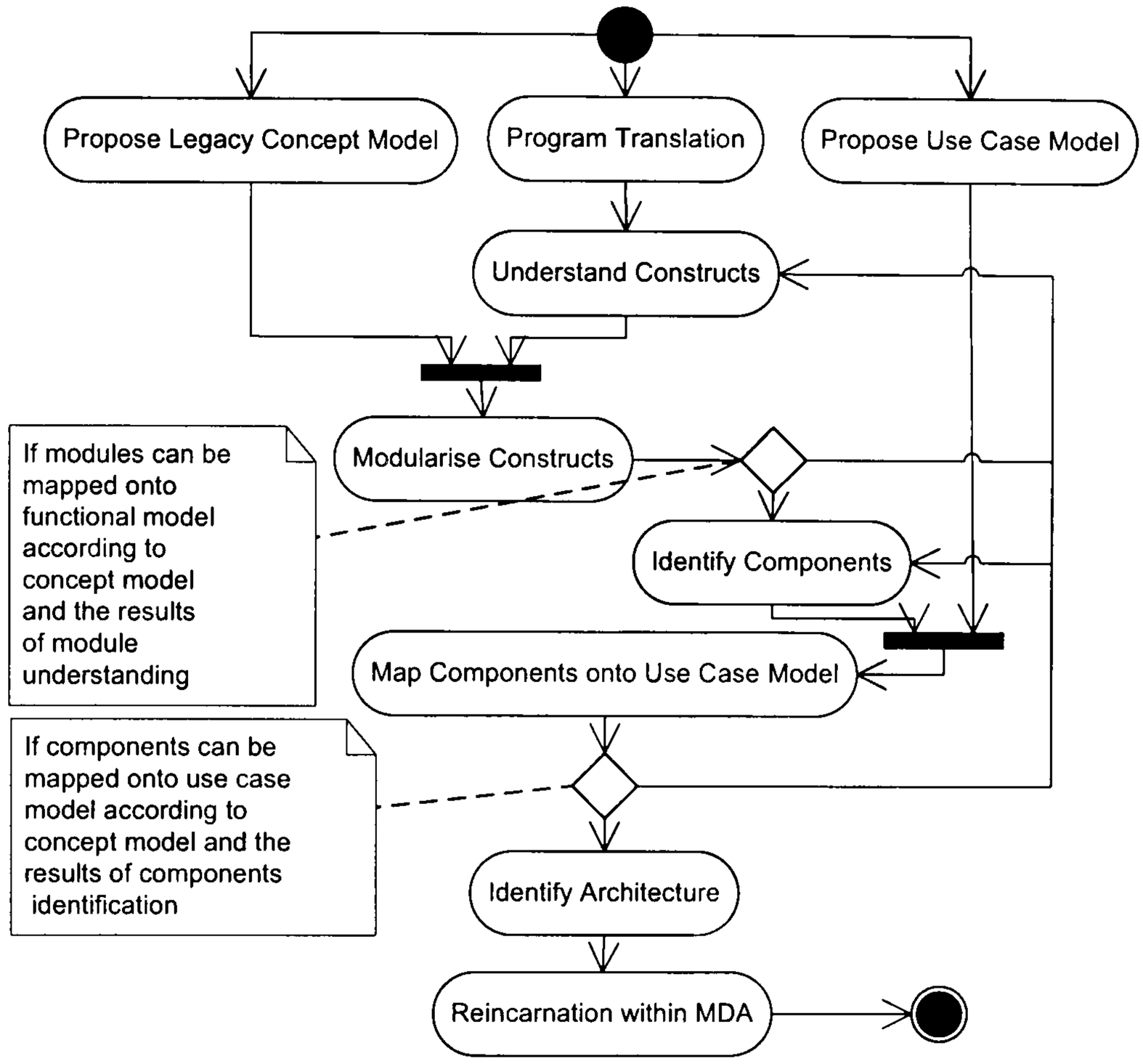

Figure 4-2-2 Unified Software Reengineering Process

A process is a series of activities bringing about the final result. Each level of the cubic model represents an activity in the abstraction process of software reengineering, which is shown in figure 4-2-2. Initial components are firstly identified in a system through system decomposition techniques. The decomposed system is then translated equivalently into a Platform Specific Model (PSM) written in a Platform Specific Language (PSL). After restructuring the PSL code, abstraction rules are applied to get abstractions of the system from constructs, modules up to the system architecture composed of interconnected components. Written in a formalised platform independent language, the extracted architecture description is a Platform Independent Model (PIM)

and can be forward engineered within MDA environment to produce new PIMs that can be later transformed into any PSMs automatically.

4.2.1 Concept and Use Case Models

- Propose Legacy Concept Model

- Propose Use Case Model

The first step is to propose a legacy concept model and a use case model that are used to make preliminary analysis on domain architecture and business objects to help identify modules and components. A thorough understanding of domain architecture patterns could facilitate the process of building the concept model. Domain patterns are discussed in Chapter 8.

A business concept model describes the real world business concepts involved in the software system. It is composed of business objects that reflect real-world objects and belong to specific classes.

- There is often (and should be) a strong parallel between a software system and its underlying business concept model: The system's software objects should parallel the enterprise's business objects. In a well-designed system, most business objects have their corresponding software objects.

- To build the concept model, a description of the structure and behaviours of the system should be produced from anyone available and involved in the design, development, operation or maintenance of this system. Then each noun in the description is examined against the following criteria.

    - Is it an object or a class of objects
    - Is it part of a problem to be solved
    - Is it an event or occurrence
    - Is it a property of an object

The nouns are then classified into various categories, which will serve as objects, classes, attributes, states, operations, events and so on in the concept model.

Sometimes the description of system's behaviours is best established as a set of outside entities or actors, playing their individual roles or use cases, which take place whenever they use the system. The use cases for a specific software system represent a use case model for this system. Use case model describes externally visible behaviours from actor's point of view, covers all scenarios at the same time and may call out some variations graphically. It is very suitable for reengineers to get a high-level overview of the software system.

Concept model and use case model together provide a starting point for cognitive reverse engineering.

4.2.2 Program Translation

- Source Code Restructuring/Refactoring

- Establish Initial Platform Specific Model

As the simplest form of reengineering, program translation converts a legacy program consistently between two or more programming languages. Source code translation often happens when porting a system written in an obsolete or proprietary language and aims to transform it equivalently into another system written in a well understood and supported language [Warren99]. Sometimes the target of program translation is not a better language but an intermediate one. One typical intermediate language is Microsoft Intermediate Language (MSIL) or IL. IL, as an implementation of the Common Intermediate Language (CIL), is the internal format of all the .NET languages. It's therefore possible to make a single compiler supporting multiple languages development environment. The concept of intermediate language is also applicable in reverse engineering where a reverse engineering approach is designed against a specific language, into which other languages can be translated via respective translators. Although source code translation will not improve the structure of the legacy program, it reduces complexity and work-load, for only one language has to be handled for

analysing a multi-language system. In this way, program translation will be the first step in understanding a legacy system.

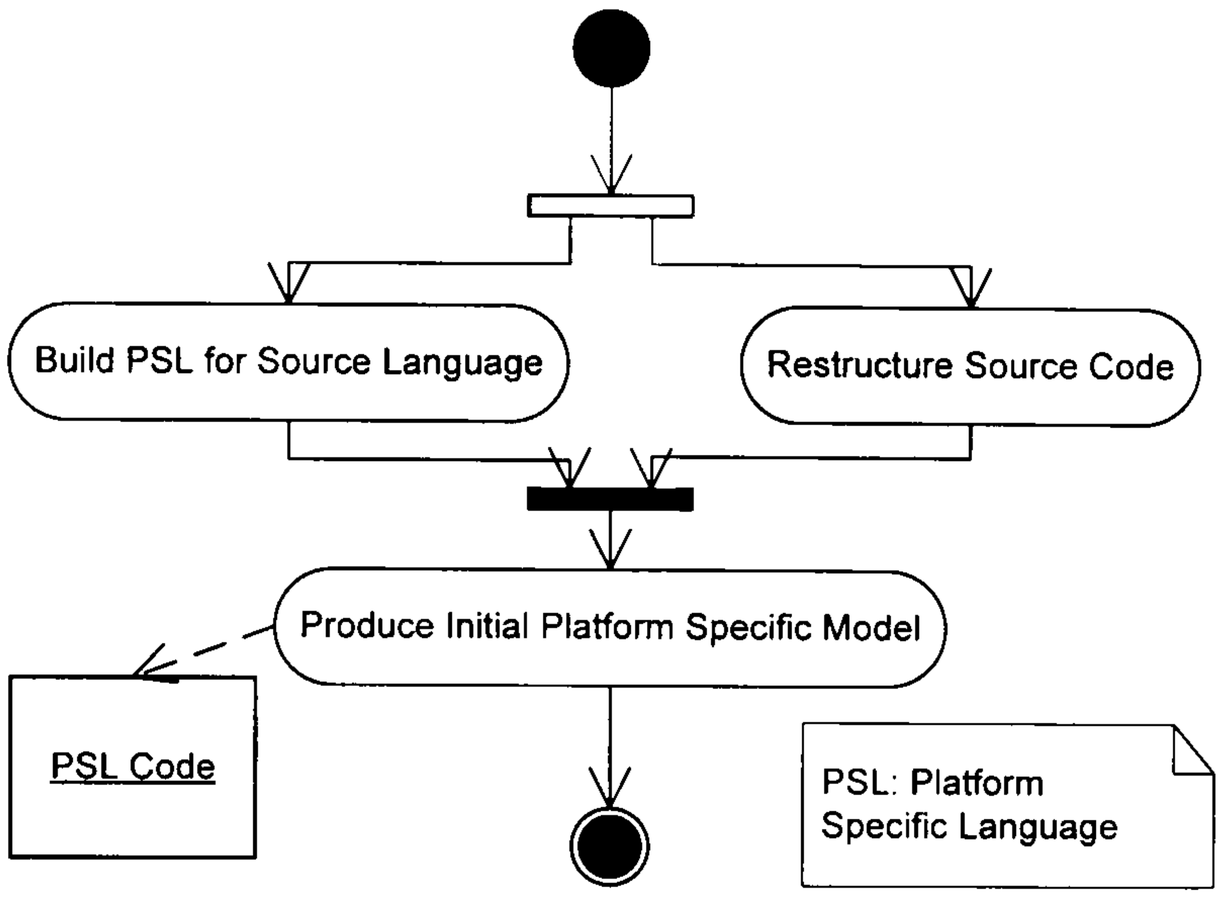

Figure 4-2-3 Process of Program Translation

The success of the abstraction process depends on a semantically equivalent program translation. All the information of the legacy system should be passed on to the initial PSM (strictly speaking, initial PSM is not a real PSM defined in MDA, for it contains all the implementation details), on which abstraction rules will be applied to produce higher level PSMs until all implementation details are left out. Such translation is achieved by defining a Platform Specific Language (PSL) for source language, which has all the counterparts of the source language. A PSL is a superset of the language used for description of its corresponding final Platform Specific Model. This facilitates maintaining consistency between initial and final PSMs after all the abstractions. Figure 4-2-3 shows the process of program translation.

In addition to translating a legacy system into PSL code, other tasks that should be done in this step include: source code restructuring or refactoring.

4.2.3 Modules Identification

- Understand Procedures

- Modularise Procedures

- Identify Concept Model

Object-oriented languages provide a well defined interface to their objects through classes that encapsulates a single design decision. As the building blocks of OO system, objects represent a higher level view than procedures in structural systems. Actually a general way to understand a structural system is to recover objects based on the relationships between variables and procedures. Therefore a well built OO system can be regarded as a modularised one, whilst a structural system has to be modularised before going through other activities of reverse engineering.

In a structural system, a complete function is implemented by a group of procedures, which are not independent on their own. For example, a procedure for returning a poster's information in a bulletin board system is called by many other procedures. Program modularisation gathers related procedures into modules, which makes it easier to identify and eliminate redundant code, optimise interactions between modules, and simplify interactions with the rest of the system [Seacord03]. Program modularisation is the second step in understanding a legacy system. Elementary abstraction rules are applied in this phase. Figure 4-2-4 shows the process of modules identification.

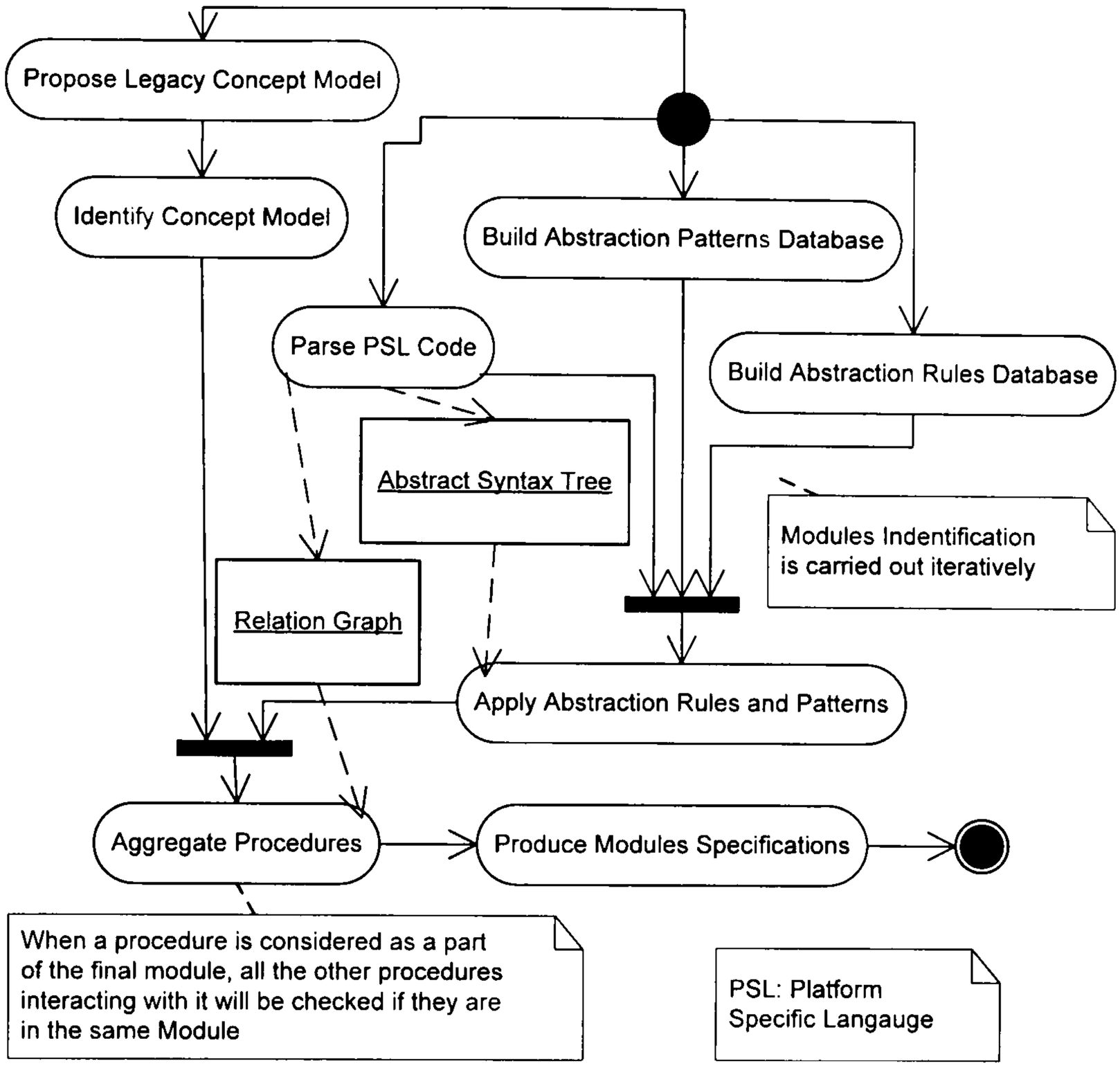

Figure 4-2-4 Process of Modules Identification

4.2.3.1 Relationships between Procedures

To find out the relationships between procedures, their internal behaviour must be analysed, which means to get a higher level view by applying a series of abstraction rules. Reverse engineering techniques are used to produce such views or specifications of individual procedures, which are then analysed together with relation graph such as call graph to aggregate related procedures into an individual module. The approach to understanding procedures of a structural system is also applicable when analysing the behaviour of an object in an OO system. Therefore only the process of understanding a structural system is depicted in figure 4-2-4. There exist many ways to modularise a legacy system. After finishing understanding individual procedures, modularisation is carried out via iteratively absorbing the leaves of the relation graph of all procedures. The level of modularisation is decided by the knowledge from concept model. For example, procedures invoking no other procedures but system APIs will be "absorbed"

into their callers, on which a further abstraction will be performed to update its specification.

4.2.3.2 Abstraction Patterns for Modules Identification

The effective application of abstraction rules is based on a successful recognition of abstraction patterns, which indicate the information that could be left out in a system representation at a higher level of abstraction. These abstraction patterns are classified into five categories in terms of their different forms in software systems: State Test and Exception Handling, User Interface Format, Semantic Core, Domain Function and Efficiency-Improving Details [Liu99].

4.2.3.3 Concept Model Identification

As stated earlier, business objects in a concept model are often tightly coupled with software objects. Some of them can be identified from source system with such obvious links as the names of attributes, classes, file names or even folders. The identified candidate representatives for the business objects can be viewed as initial modules.

4.2.4 Components Identification

- Identify Components
- Map Components onto Use Case Model
- Decompose Use Case Model

Although modularisation provides a higher level of abstraction than the original system, the produced modules are still fine-grained and tight-coupled due to the lack of design concerns during aggregating procedures. To address this issue, in the phase of components identification, top-down and bottom-up strategies are combined to recover the relationships between modules and thus aggregate them into components. Figure 4-2-5 shows the process of components identification.

The process of identifying design concepts includes: identifying encapsulated data and databases, identifying data and control flow, identifying encapsulated functions and clustering the identified modules. The final step, clustering the identified modules, will

produce components specifications based on modules identification. Components exist at a higher level of abstraction than modules or objects. Components are coarse-grained and have a wide range of intercommunication mechanisms with strictly defined interfaces.

When identifying components from a legacy system, both bottom-up and top-down strategies are adopted. Whilst individual components are understood by analysing the modules aggregated in them, the interactions between components need to be mapped onto a concept model proposed by domain experts. Included in such a model are design patterns that likely exist in the legacy systems. The produced components specifications are then used as the initial building blocks of the legacy system architecture. Architecture abstraction rules are used in this phase.

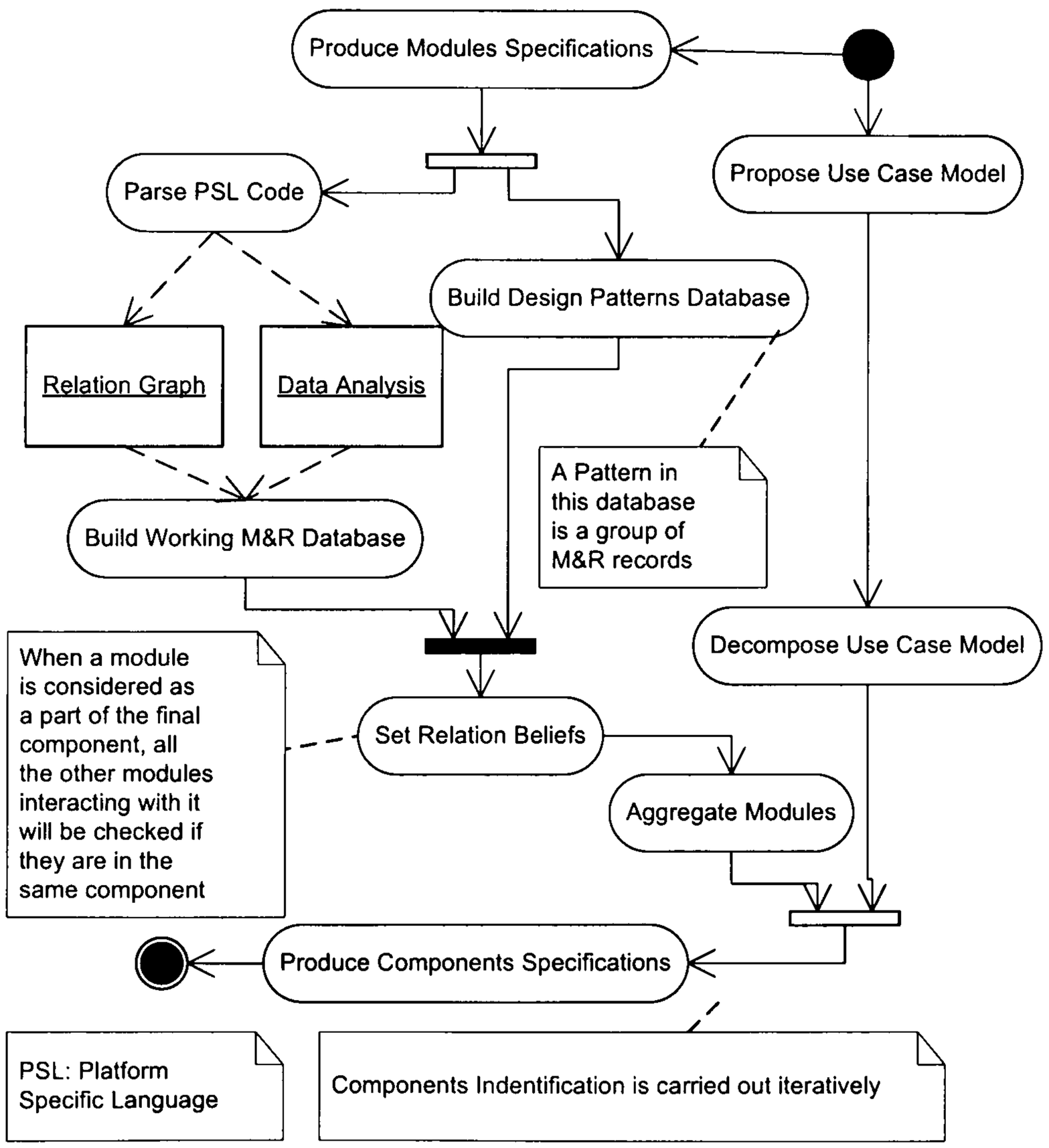

Figure 4-2-5 Process of Components Identification

4.2.4.1 Relationships between Modules

In the phase of modules identification, relation graph is used for aggregating procedures by absorbing the leaves into callers. This time, however, relation graph will be combined with data analysis to build a Working Modules & Relations database that has tables with four columns: "Module A", "Module B", "Relation" and "Belief". Whilst "Module A" and "Module B" are input for module aggregation, the "Relation" represents the possible relationship between them within a certain design pattern and the "Belief" denotes the possibility of this relationship.

Setting beliefs requires handling uncertainty, where a number of factors, including procedure calls, global variables and data accessing, could affect a belief value. An appropriate uncertainty description is therefore necessary to address this issue. Certainty Factor overcomes the heaviness of other uncertainty descriptions, such as Machine Learning techniques represented by Artificial Neural Networks, Non-monotonic Descriptions represented by Default Reasoning and Minimalist Reasoning, and Statistics-based uncertainty description represented by Bayesian Networks [Li02]. To balance the accuracy and complexity, Certainty Factor is employed in this phase to set beliefs for relations between modules in respect of a possible design pattern.

Abstractions in the phase of modules identification can help understand individual modules but in the meantime less relevant statements are left out. To be complete and accurate, the relation graph for component identification must be constructed from the source code rather than abstracted modules.

4.2.4.2 Design Patterns for Components Identification

Design pattern describes a commonly recurring structure of communicating components that solves a general design problem within a particular context. Patterns are usually described using a format including the following information:

- A description of the problem that includes a concrete example and a solution specific to the concrete problem

- A summary of the forces that lead to the formulation of a general solution

- A general solution
- The consequences, good and bad, of using the given solution to solve a problem
- A list of related patterns

Design patterns are represented as a group of M&R records with predefined beliefs. During the process of identifying components, a design pattern is matched against the Working M&R database. Every successful matching between the records of them may lead to revision to the beliefs of the relevant records in the Working M&R database. A design pattern will be recovered when the related beliefs reach a predefined level. At this point, all the modules participating in this design pattern can be grouped by their respective roles into components. For example, when an abstracted module with a given belief is considered to be inside a component that plays a role in a design pattern, such as a client in client/server architecture, all the other modules interacting with it will be checked on whether they are inside the same component and their beliefs will be revised accordingly.

4.2.4.3 Use Case Model Decomposition

When building use case model, the use case diagrams are drawn for not only the whole system, but also any of its complex subject that needs to be treated as blackbox that specifies external and visible behaviour but hides the internal one.

In larger systems, such a whole picture needs to be decomposed into small ones for corresponding subsystems. These subsystems can be treated as use case subjects. Their actors will be those entities that are external to each subsystem.

4.2.5 Architecture Recovery

- Identify Architecture
- Produce Final Platform Specific Model
- Produce Platform Independent Model

- Reincarnation within MDA

Architecture recovery is an interpretive, interactive, and iterative process, which requires the efforts of both reverse engineering experts and architects so that architectural constructs can be recovered by analysing diverse mechanisms in an implementation [Li02]. Architecture recovery aims to produce a highest level of abstraction of a legacy system, which is built on the understanding of individual procedures, modules and components. An appropriate description language is required to represent the recovered architecture, for such a high level abstraction is not only for developers, but managers and other stakeholders. The initial architecture, made up of identified components, is matched against an architectural pattern proposed by domain experts. This pattern gives a clue to the search of candidate architecture, whilst the identified components are used to update the initial architecture by interactions between components.

An identified component represents a functional, relatively independent part of the legacy system that is the building block for the final architecture description. Similar to components identification, architecture recovery is performed by combining top-down and bottom-up strategies, i.e., analysing the relationships between components and their connectors against architectural patterns proposed by domain experts. After abstractions leave no more implementation details in the working PSM, it becomes the final PSM, from which the Platform Independent Model will be produced independent of the specific platform technology. As PIM contains neither platform, nor implementation details, it can be transformed into PSMs by any tools compliant with MDA. Figure 4-2-6 shows the process of architecture recovery. Architecture abstraction rules are used in this phase.

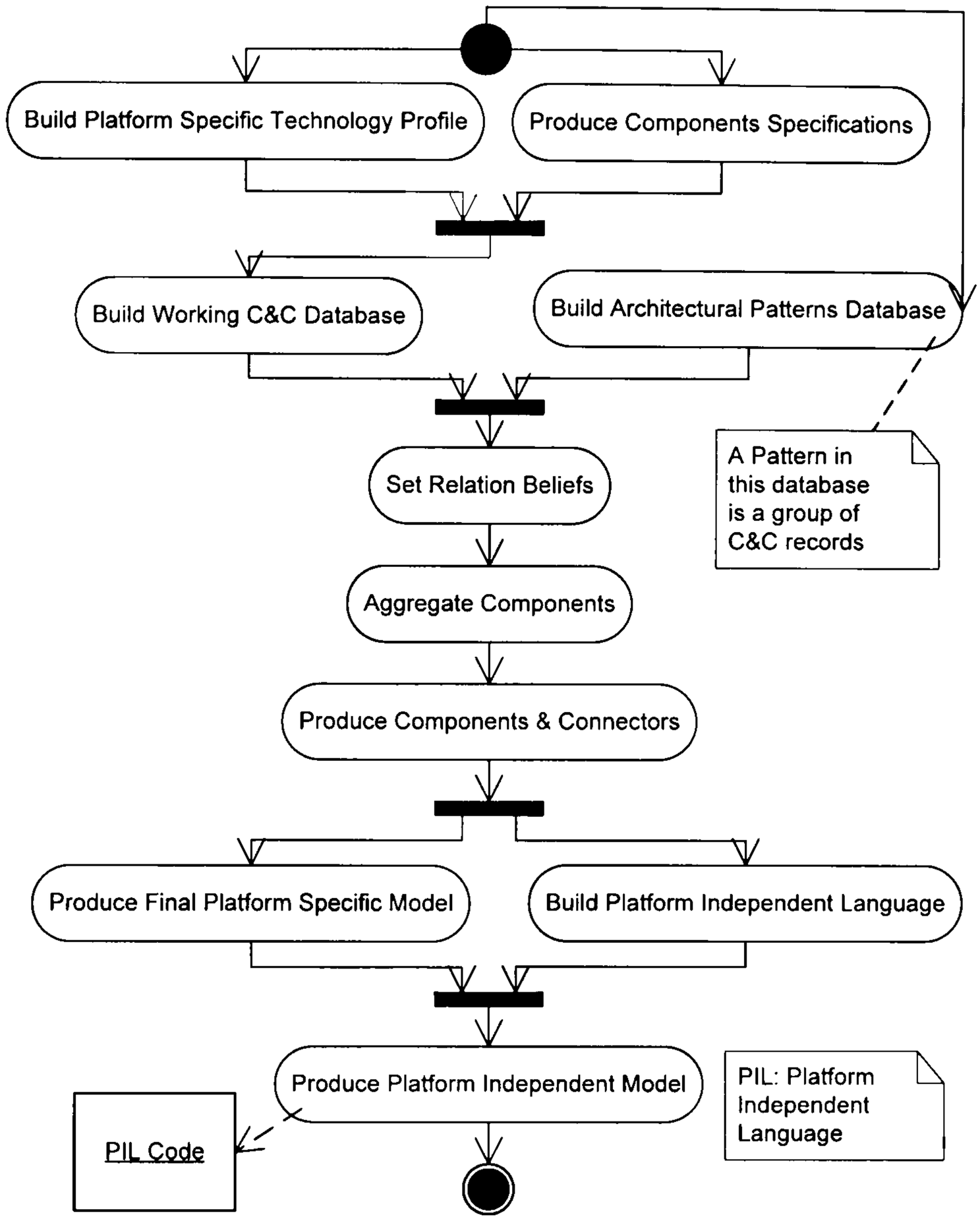

Figure 4-2-6 Process of Architecture Recovery

4.2.5.1 Relationships between Components

Relationship analysis in this phase is still based on relation graph of components produced from components identification. Relation graph will be combined with components specifications to build a Working Components & Connectors database that has tables with four columns: "Component A", "Component B", "Connector" and "Belief". While "Component A" and "Component B" are input for component aggregation, the "Connector" represents the possible connection between them within a certain architectural pattern and the "Belief" denotes the possibility of this connection.

Setting beliefs for C&C database is similar to that for M&R database where Certainty Factor is employed to handle uncertainty.

4.2.5.2 Architectural Patterns for Architecture Recovery

An architectural pattern provides a system template in a specific domain. A specific component of an architectural pattern typically can be mapped onto a certain part in source code that in turn might be mapped onto one or more design patterns. Architectural patterns are divided into four categories by the characteristics of the systems in which they are most applicable:

- Structure: Layers, Pipes and Filters, and Blackboard
- Distributed Systems: Broker
- Interactive Systems: Model/View/Controller and Presentation/Abstraction/Control
- Adaptable Systems: Reflection Microkernel

Such classification can be more detailed. For example, a web-based system is only one form of distributed systems. Architectural patterns are represented as a group of C&C records with predefined beliefs. During the process of architecture recovery, an architectural pattern is matched against the Working C&C database. Every successful matching between the records of them may lead to revision to the beliefs of the relevant records in the Working C&C database. An architectural pattern will be recovered when the related beliefs reach a predefined level. At this point, all the components participating in this architectural pattern can be grouped by their respective roles to produce a complete architecture description.

4.2.6 MDA Models Identification

The PIM produced from Architecture Recovery step needs to be refined before feeding into MDA tools for automatic transformation. This process involves four main activities that should take place in an iterative and incremental fashion.

- Analyse initial PIM and collect feedback. Every domain subject has a set of rules placed upon it by its shareholders. The refined PIM should match those rules as close as possible. Therefore, analysing initial PIM against those rules and collect feedback from shareholders should be carried out to understand the inconsistencies between the two before refining the model. There are many ways to facilitate this work, such as use case diagrams, formalised documents or simply text.

- Abstract Sample PIM. Given a set of rules, there are many ways of abstracting a sample PIM to the initial one. Abstraction therefore involves analysing all available information, either out of reengineering or shareholders, and building the sample model in a particular way.

- Refine PIM. The sample PIM describes and defines the abstracted rules as a formalisation of knowledge of the subject matter. Once a proper sample PIM is produced, the next step is to refine the initial PIM produced from previous steps against the sample PIM.

- Test Model. The PIM is verified in every time of the refining iteration to guarantee that the model is correct, i.e., whether the PIM meets the expectation of the reengineers or the shareholders?

Figure 4-2-7 shows the process of Model Identification.

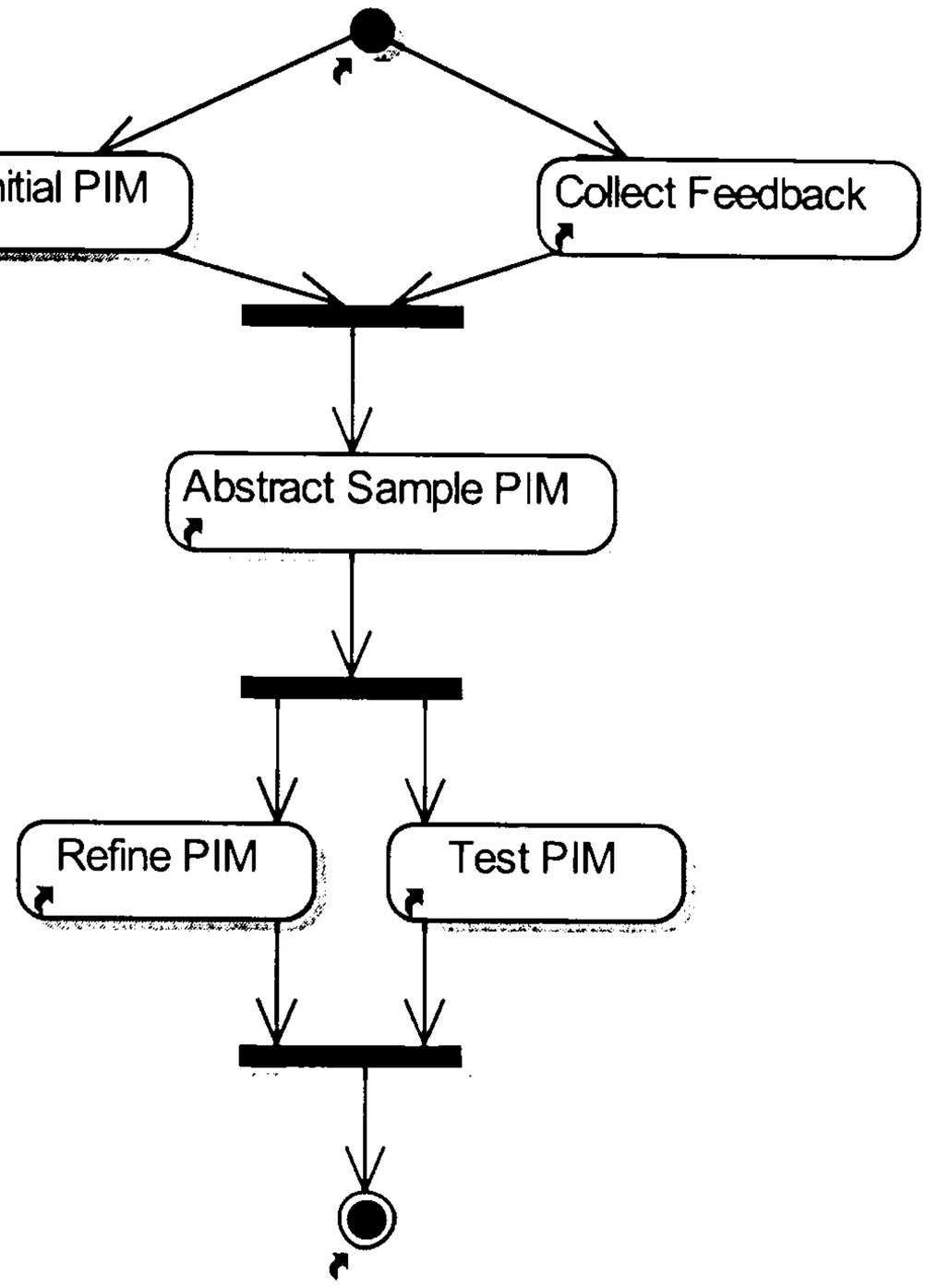

Figure 4-2-7 Process of Model Identification

4.2.7 Architecture Migration

- Gather new requirements
- Choose Domain Architectural Patterns
- Improve Platform Independent Model
- Produce Platform Specific Model
- Produce Implementation

The process of architecture migration is shown in Figure 4-2-8. Architecture migration is to engineer a new architecture from the existing one. It can be regarded as a kind of forward engineering where a new system is built with the help of architecture recovery results.

In addition to software architecting experience including usage of architectural patterns, domain and system knowledge is required to define a proper architecture. Reverse engineering not only helps in extracting domain knowledge from the system, but clarifying existing architectural patterns in the system. The recovered patterns may influence decisions made during architecture migration. Domain patterns are discussed in 8.1.

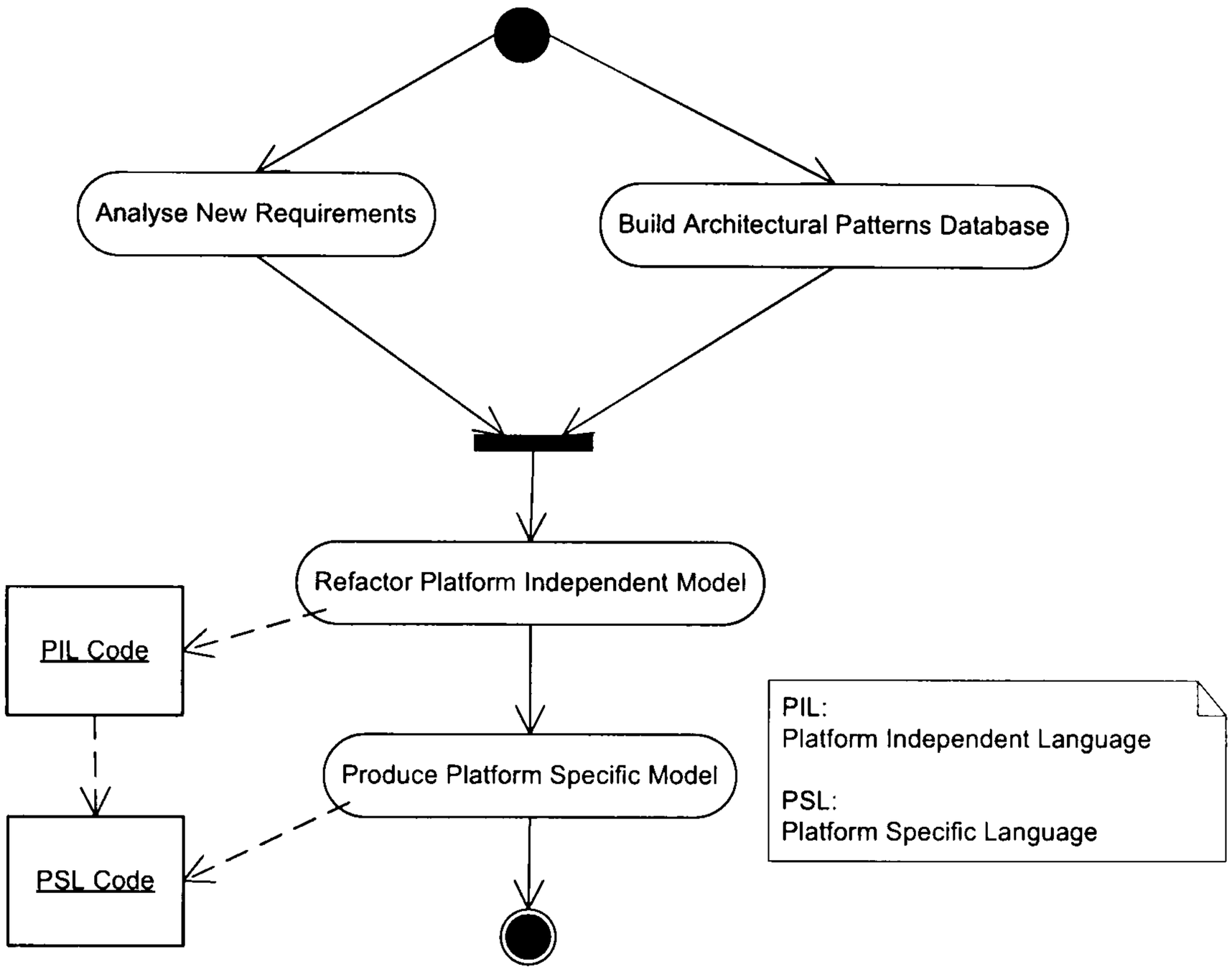

Figure 4-2-8 Process of Architecture Migration

Migrating from PIM to PSM consists of four steps.

- Specify Mapping Functions. The source model, refined PIM, is transformed into a target model that captures the abstractions implementing the concepts of the source model. A mapping function for such transformations should be extensible and expressive so that the concepts of MDA can be applied to the specification of mapping functions to transform models into executable implementations.

- Mark the Models. Each model element in the source model is marked to indicate that a particular mapping function must be used to determine the kind of associated target element. If there is a mapping function for every element of the source metamodel, and a mark or default rule for unambiguously selecting the mapping function for every element of the source model, the marking models can be established completely.

- Verify the Mappings. After finishing the PIM and marking the model with defined mapping functions, the combination of models and marks should be verified to guarantee valid results of transformations. Mapping functions have to account for some source model constructs that cannot be translated plainly into target model constructs. The path of mappings from the most abstract to the implementation-oriented metamodels must be maintained to avoid losing formalised knowledge in the source model.

- Transform the Models. Model transformation is anticlimactic, given they have been marked and the mapping function specifications are complete.

4.3 Summary

This chapter summarises the concepts, architecture and building blocks of Web-based Systems Evolution.

- A round-trip engineering is introduced, where adaptable MDA (AMDA) is recommended as the viable approach to software development and reengineering.

- Being of significant importance to both software development and reengineering, the concept and taxonomy of abstraction are discussed and will be given more details in Chapter 6.

- A Web Application Infrastructure for building evolvable applications is presented. A Web Application Framework based on this infrastructure will be introduced in Chapter 5.

- A Reengineering Framework focusing on reengineering existing or legacy systems to evolvable systems is defined.

- Finally, the whole process of software evolution is elaborated.

The rest of this thesis will focus on individual components that make up the framework, process and architecture presented in this chapter.

CHAPTER 5

Web Application Infrastructure and Framework for Evolvable Web-based Systems

As discussed in related work, current Web Application Frameworks (WAF) do not promote good development practices and applications built on them tend to interpose container or platform dependencies into business logic, which presents a major barrier to software maintenance and reengineering. In addition, the would-be systems resulting from reengineered legacy systems should be built on a framework that is evolvable and can be better maintained. This chapter introduces the proposed Web Application Infrastructure and Framework for building evolvable Web-based systems.

5.1 Design Issues of Evolvable Applications

A sound understanding on designing evolvable applications is essential in building a good Web application, the supporting Web Application Infrastructure and Framework.

5.1.1 Inheritance and Interface in OO

As the most important mechanisms of OO programming, the way inheritance and interface are used has great impact on designing evolvable applications.

- Loose Coupling between Objects. Loose coupling between objects requires programming to interfaces that decouples interfaces from their implementations and thus promotes flexibility to change the implementing class of any object without affecting the calling code, which is particularly important in Web-based systems due to their scale.

- Object Composition and Concrete Inheritance. Object composition is more flexible than concrete inheritance. It allows the behaviour of an object to be altered at run time by delegating part of its behaviour to an interface, of which the implementation can be set by callers. Concrete inheritance provides

polymorphism and can be implemented more conveniently with code inherited from a superclass.

5.1.2 Design Patterns for Building Evolvable Applications

Design patterns [Gamma95], such as Template Method, Strategy, Callback and Observer, can be used to build application infrastructure that decouples components and enables extensibility with modification.

- Template Method design pattern uses an abstract superclass to encapsulate individual steps as abstract methods and invoke them in the correct order to control the workflow. Concrete subclasses of this abstract superclass implement the abstract methods performing the individual steps. Such uses of the Template Method pattern offer good separation of concerns, where the superclass concentrates on business logic, while the subclasses on implementing primitive operations. The Template Method design pattern is especially valuable in infrastructure design. Code example for Template Method pattern can be found in Appendix C.

- The Strategy Design Pattern factors the variant steps into an interface, of which the implementation is used as a helper by a concrete class to construct a workflow. The Strategy design pattern is a bit more complex than the Template Method pattern, but does not force the class that implements the individual steps to inherit from an abstract template superclass. Code example for Strategy pattern can be found in Appendix C.

- Callback Pattern, a special case of Strategy design pattern, can parameterise a single operation, while moving control and error handling into a framework. It is based around the use of one or more callback methods invoked by a method that constructs a workflow. Code example for Callback pattern can be found in Appendix C.

- Observer Design Pattern can be used to decouple components and enable extensibility without modification. The proposed Web Application Infrastructure provides an event publication mechanism, allowing good separation of concerns

without the need for an application to implement any plumbing code. Code example for Observer pattern can be found in Appendix C.

5.1.3 Application Registry for Building Evolvable Applications

The Singleton design pattern is widely used in Web applications. A typical implementation of a singleton in Java includes a static instance variable to hold the singleton instance, a public static method to return the singleton instance and a private constructor to prevent instantiation, as shown in Listing 5-1-1.

```
public class TypicalSingleton {

  private static TypicalSingleton instance;

  static {
     instance = new TypicalSingleton();
  }

  public static TypicalSingleton getInstance() {
     return instance;
  }

  private TypicalSingleton() {
          ...
  }
}
```

Listing 5-1-1 Implementation of Singleton in Java

There are many drawbacks in using singletons despite its popularity. Singleton keeps the initialisation process to itself and thus must handle its own configuration, loading any properties required. Dependence on the singleton class is hard-coded into many other classes. It is also difficult to manage singletons in complex applications, as they are scattered throughout the code and tend to have their own configuration loading method. Finally, singletons are not meant to be used for either interface or inheritance, as they are bound to the singleton class and rely on static variable.

As an alternative to singleton, a registry object accessible to all relevant objects can be used to hold indices to application resources. An object in the application only needs a reference to the single instance of the registry object to retrieve the single instances of any application object. In the case of J2EE, this registry object can be held in the Web application's ServletContext and does not need to be a singleton. The registry object is

part of the proposed infrastructure code used by multiple applications. Listing 5-1-2 shows the use of registry object in J2EE Web application, where the registry object loads configuration and registers itself with the ServletContext of the Web application.

```
// Objects needing to use "singletons" must look up the context object in:
 ApplicationRegistry appReg = (ApplicationRegistry )
 servletContext.getAttribute("org.myapp.registry.ApplicationRegistry");

// The ApplicationRegistry instance can be used to obtain any "singleton"
 TypicalSingleton typicalSingleton = (TypicalSingleton )
 appReg.getSingleInstance("typicalsingleton");
```

Listing 5-1-2 Registry Object for Evolvable Application

The registry object has some advantages. It works well with interfaces and inheritance. It is responsible for instantiating and configuring individual singletons, which means that configuration outside Java code can be used to load data. Javabeans allow easy property discovery and manipulation at runtime. Application objects designed as JavaBeans can be instantiated and configured easily using configuration data outside Java code.

5.2 Web Application Infrastructure

An infrastructure specifies a collection of built-in system services and configuration mechanisms that provides a running environment for software systems. A good infrastructure contributes significantly to the success of software development and reengineering. This section introduces the proposed Web Application Infrastructure.

Component-based software development is the nowadays dominant software technology and has greatly improved the development and maintainability of software systems. Most Component-based Infrastructures aim at two essential goals to achieve evolvable software systems: separation of concerns and externalisation of responsibilities from source code. However, the work is only done partly to date.

5.2.1 Design Issues of Web Application Infrastructure

The core concept of Component-based infrastructure is container that provides application code a running environment, i.e., a collection of services and management

mechanisms. A container that meets the two goals given above should meet the following requirements:

- Lookup. The container should act as a factory in providing references to managed objects by separating calling code from implementation details that is hidden inside the container.

- Configuration. The container should support a consistent and parameterised configuration mechanism for objects running within it. Configuration values should be externalised as much as possible from source code to avoid recompilation due to changes to them.

- Dependency Management. A container should be able to manage relationships between objects. Such relationships should be externalised as much as possible from source code for the same reason as configuration of objects.

An evolvable container should have the following features:

- A container should not impose special dependencies on the managed code, which in turn should be container independent. This feature enables legacy code to be run inside a container without modification.

- A container should have minimal API dependencies to be run in a variety of environments, such as web container, standalone client, or even an applet. This feature makes it easy to reengineering various parts of legacy systems as required.

- A container should minimise the deployment effort and performance overhead for adding a managed object to enable managing fine-grained objects.

An evolvable container is very different from existing ones. For example, an EJB (Enterprise Javabeans) container does not meet the criteria set above:

- Code written to the EJB model can not run without the EJB container.

- EJB deployment involves multiple Java source files per EJB and possibly a code-generation and compilation step.

- Code is written to run in a specific environment.

- EJBs are not suited for fine-grained objects.

As a container, EJB model does not support managing relationships between managed EJBs that has to use JNDI (Java Naming and Directory Interface) to reference other EJBs. It is verbose to configure even simple properties, where bean implementation code has to use JNDI to look up untyped "environment variables" defined in lengthy XML deployment descriptors.

5.2.2 Evolvable Web Application Infrastructure

The proposed infrastructure is central to solving the design issues of building evolvable applications discussed in Section 5.1. It uses the concept of Dependency Push and JavaBeans to minimise the dependence of application code on it, achieving maximum loose-coupling. This section examines the support packages used in the case study in Section 11.3.

5.2.2.1 Container of Web Application Infrastructure

An evolvable infrastructure must eliminate the dependency of applications on it. In real world, only very simple objects work in isolation, while most business objects have dependencies on other business objects, data access objects and resources. Such dependencies need a proper lookup mechanism that should not, however, introduce a dependency on a container.

As stated earlier, the container is the core of a Component-based infrastructure. An evolvable container can be implemented in one of the two main ways:

- Dependency Pull. The container provides callbacks to components, and a lookup context, which is the EJB and Apache Avalon approach. Each component needs to use container APIs to "pull" resources and collaborators on its own.

- Dependency Push. The container provides dependency resolution, wholly responsible for wiring up components with resolved objects "pushed" in via properties or constructor arguments.

Figure 5-2-1 illustrates the two kinds of evolvable containers.

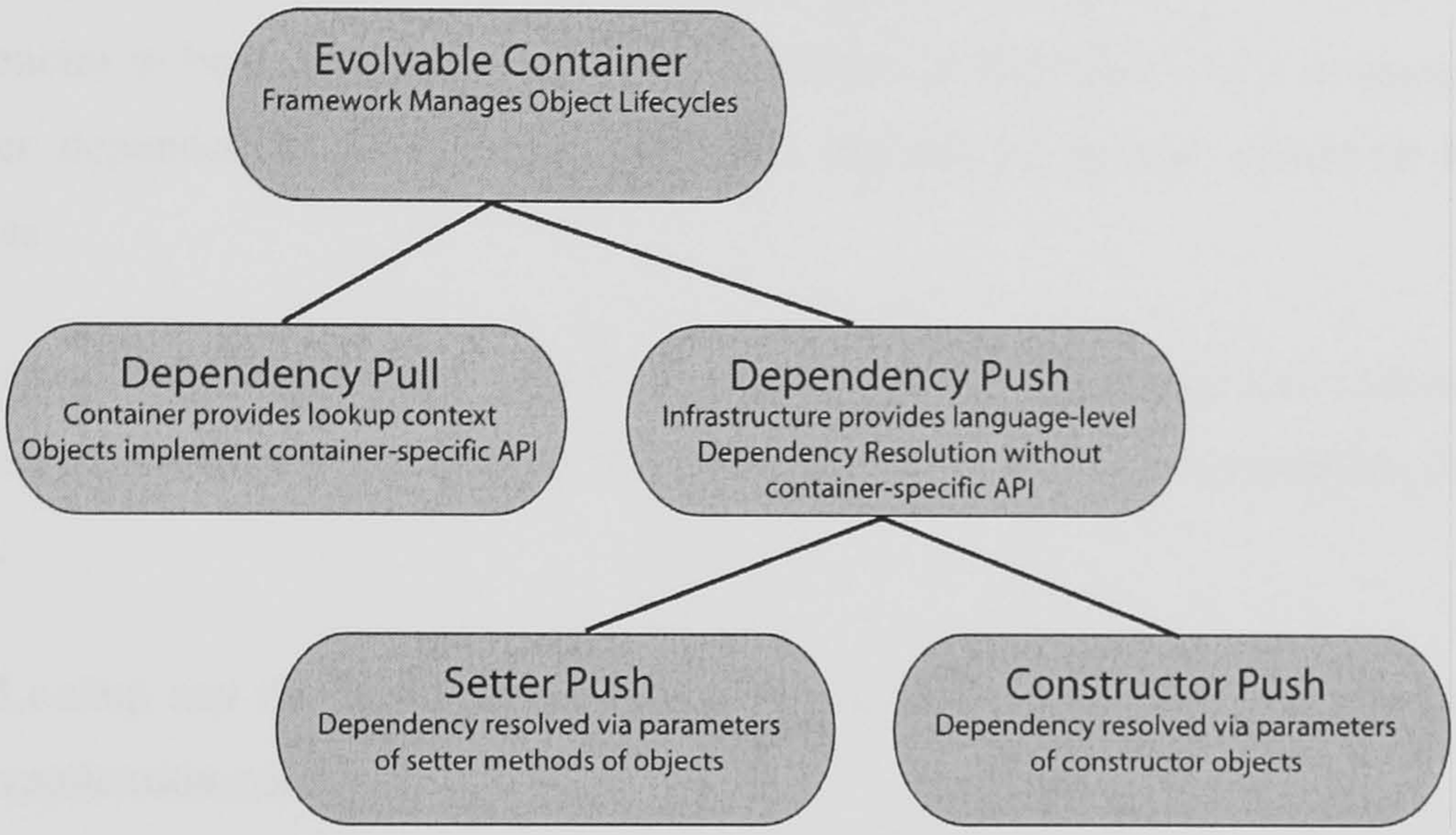

Figure 5-2-1 Hierarchy of Evolvable Containers

Most popular containers of Component-based infrastructures, such as EJB, provide the Dependency Push mechanism, where the container manages object lifecycle and the managed objects are responsible for lookups on their own.

An EJB references other EJBs and resources via JNDI that decouples this object from the implementing DataSource and collaborator, and externalises its properties. A different data source or interface implementation can be used without changing Java code, so pluggability is achieved and the code is portable between application servers supporting standard J2EE APIs.

However, such containers have inherent disadvantages over developing evolvable applications shown as follows. Firstly, the class must run inside an application server environment for its dependency on JNDI that is hard to decouple from an application server and thus makes any structural improvements very difficult. To use something other than JNDI to look up resources and collaborators, the JNDI lookup code has to be re-factored into a method that could be overridden, or a Strategy interface. Secondly,

this class is hard to test during both development and reengineering processes. A dummy JNDI context must be provided to do effective unit testing.

The second container implementation strategy, Dependency Push, is more preferable in terms of building an evolvable infrastructure, where the container is responsible for dependency lookup, and the managed object expose their properties setup to enable dependencies to be passed into it when the container is initialised. As a language-based container dependency resolution, it does not depend on special container APIs or interfaces.

With Dependency Push, the container is responsible for looking up the resources, and providing the necessary resources to the business object, which offers benefits shown as follows:

- Lookup and the related dependency on container are completely removed from application code.

- Without dependency on a container API, only language concepts exist in application objects that can be tested outside any container or developed/reengineered without the hassle of decoupling container specific details.

This implementation strategy minimises dependency of application code on the container and can be applied for collaborating objects as well as simple resources such as Strings and ints.

Dependency Push strategy can be implemented in two ways: Setter Dependency Push and Constructor Dependency Push.

With Setter Injection, components expose dependencies on configuration values and collaborating objects via properties based on a language-specific convention. A Setter Dependency Push example in Java is shown in Listing 5-2-1.

```
public class BusinessObject implements BusinessInterface {
  private DataSource ds;
  private Collaborator collaborator;
```

```
  private int intValue;
  public void setDataSource(DataSource ds) {
  this.ds = ds;
}

public void setMyCollaborator(Collaborator collaborator) {
  this.collaborator = collaborator;
}

public void setMyIntValue(int intValue) {
  this.intValue = intValue;
}
}
```

Listing 5-2-1 Setter Dependency Push

The setter methods are invoked immediately after the object is instantiated by the container, before it handles any business objects that no longer need any resource lookup and dependency on JNDI.

With Constructor Dependency Push, components expose dependencies via constructor arguments. A constructor Dependency Push example in Java is shown in Listing 5-2-2.

```
public class BusinessObject implements BusinessInterface {
  private DataSource ds;
  private Collaborator collaborator;
  private int intValue;
  public BusinessObject(DataSource ds, Collaborator collaborator, int intValue) {
  this.ds = ds;
  this.collaborator = collaborator;
  this.intValue = intValue;
 }
}
```

Listing 5-2-2 Constructor Dependency Push

5.2.2.2 Implementation of Dependency Push Container

An evolvable application needs a consistent way to handle configuration that addresses the design issues discussed in Section 5.1. A Dependency-Push container is the appropriate way to provide such consistency and reduce complexity of application code:

- Consistency. Configuration management could be haphazard without being supported in infrastructure. The Configuration of J2EE Web applications could appear in such a variety of formats as properties files, XML documents, JAR

(Java Archive) and WAR (Web Archive) development descriptors, and database tables. Their management is often in little consistency. In addition, decentralised configuration management leads to applications adopting a variety of approaches to looking up application objects, resulting in problems such as overuse of Singleton design pattern, tight-coupling to JNDI and global ServletContext of Web applications.

- Complexity. It is difficult to achieve, without infrastructure support, configuration externalisation that is the key to the essential issue of designing evolvable applications, application parameterisation. Application classes would have to handle configuration management code irrelevant to their responsibilities in domain model. Their business logic is obscured and they are tightly-coupled to the configuration format such as XML document.

5.2.2.2.1 JavaBeans for Configuration Externalisation

Bean-based manipulation has many advantages in handling configuration externalisation and data binding, such as from HTTP requests onto Java object state in Web applications.

- JavaBeans maximise the ability to separate configuration data from application code.

- Bean-based application components can be configured in a consistent way irrespective of the form of the configuration data.

- All that is required for a simple JavaBean is a no-argument constructor and property methods following a naming convention, without being forced to implement any special interfaces or extend a special superclass.

However, the core JavaBeans API does not support some useful operations, such as combined exception for setting multiple properties in a single operation, bean event propagation without complicating the implementation of JavaBeans and a standard method to perform initialisation on a bean after all its properties have been set. The lowest layer of the proposed infrastructure is the org.evolvable.beans package

enhancing bean-based manipulation with the above functionality. The details of the enhanced bean-based manipulation are given in Appendix C.

5.2.2.2.2 Bean Factory for Higher Level Operations

The org.evolvable.beans package is not suitable for high-level operations. It is, however, a building block, on which a higher level of abstraction can be constructed to conceal the low-level details. The org.evolvable.beans.factory package provides a way of obtaining beans by name from a central configuration repository via a "bean factory" that exempts individual Java objects from reading configuration properties or instantiating objects. Configuration data is instead retrieved by the bean factory of the infrastructure to set the bean properties exposed by each application object. The factory is capable of providing instances of beans with unique names and constructing sophisticated object graphs via references between beans within the same factory. The details of bean factory are given in Appendix C.

5.2.2.2.3 Application Registry

An application registry is built on the bean factory and provides look-up mechanism for JavaBeans of an application. The org.evolvable.registry.ApplicationRegistry interface provides ability to publish events using the Observer design pattern, participate in a hierarchy of application registries, share objects between application components, look up messages by string name, facilitate testing and provide consistent configuration management in different types of applications.

5.3 Web Application Framework

It is vital to choose an evolvable WAF in the beginning of a new development project, for the ability of a Web application to accommodate changes depends largely on the framework it was built on. A successful reengineering project also needs such a WAF as the target of the forward engineering, where the amount of effort spent on reverse engineering should never be required again once the legacy application was successfully transformed into an evolvable WAF.

5.3.1 Design Issues of Web Application Framework

An evolvable WAF should have a clean and thin Web tier:

- A clean web tier separates control flow and the invocation of business objects (handled by Java objects) from presentation (handled by view components such as JSP pages). Java classes should be used to control flow and initiate business logic. Java classes should not be used to generate markup. It should be possible to change presentation markup without modifying Java classes. JSP pages - or whatever templates are used - will contain only markup and simple presentation logic. This permits an almost complete separation of Java developer and markup author roles.

- It is also vital to ensure that the web tier is as thin as possible. This means that the web tier should contain no more Java code than is necessary to initiate business processing from user actions, and display the results. It means that the web tier should contain only web-specific control logic (such as the choice of the layout data should be displayed in) and not business logic (such as data retrieval).

The first requirement, a clean Web tier, complies with the concept of Separation of Concerns. The concern of Control flow and business logic should be separated from that of presentation. The second requirement, a thin Web tier, concerns more about the practice of development and reengineering. Applications with business logic tied to the Web tier are hard to develop, maintain and evolve for the following reasons:

- Hard to test for both development and reengineering. For applications with well-defined interfaces for business objects, it is relatively simple to test the web interface by checking that the interface maps user actions onto corresponding business requests and displays the results in a correct way. In addition, testing business logic is carried out in isolation from any user interface. For applications with business logic in web tier, the business logic and the web interface must always be tested as a whole, which is likely to be much harder.

- Hard to reuse business logic code. This is more relevant to development, where code with container dependency often won't be reusable even to other parts of the same web application.

- Hard to change presentation or web interface workflow. A distinct layer of business interfaces facilitates regression tests which are important for both development and reengineering. Tests for Web tier bundled with business logic will need to be redesigned to reflect any changes made to workflow.

- Hard to extend. Business operations exposed in a layer of business interfaces can switch between different implementations without affecting calling code. This is very difficult, however, for business operations tied to the web tier.

A clean Web tier can be enforced by a well designed MVC framework. A thin web tier, though as important as a clean web tier, is often ignored by software developers and reengineers, where the relationship between web tier and business objects is not designed for better maintainability.

5.3.2 Model, View and Controller for Evolvable Web Framework

5.3.2.1 MVC Variants

The implementations of MVC are divided into two categories depending on the way that models and views communicate:

- In MVC architectures for standalone applications, such as that of Java's Swing, the model pushes notifications to any number of listeners, which are typically views. This is called a push model.

- In MVC architectures for Web applications, the request-response pattern of Web implies that changes in a web application can not be reflected to the user unless a request is received and a new page is generated. To implement MVC for Web applications, a view needs to have access to the models required to render a dynamic page. The view itself is responsible for pulling data from the model. This is called a pull model.

Both models deliver key benefits of MVC, where each component type has a clear responsibility:

- Models have no view-specific code.

- Views have no control or data-access code and are dedicated to displaying data.

- Controllers are responsible for creating and updating models, independent from view implementations.

5.3.2.2 Controller

A web application controller needs to fulfil key responsibilities as follows:

- Analyse request parameters. The values of request parameters are passed to business objects in appropriate forms.

- Delegate request to business service layer. Only controller objects have easy access to application business service layer, which ensure a clean and thin Web tier.

- Build and pass to views data models based on the results of business processing.

The responsibilities fall into two categories: application functionality, such as delegation to business service layer and plumbing, such as mapping models to views. The former is usually implemented by application-specific code, while the latter by generic libraries of MVC framework.

5.3.2.3 Model

The sole purpose of a model is to provide data for related views to display. In Web applications (as opposed to standalone applications), models are usually implemented as dumb storage objects, e.g., value objects, holding the result of a complete business operation. Often implemented as domain objects usable outside the Web tier, it should not have any dependency on data resources, business objects or specific Web application framework.

5.3.2.4 View

A view renders the information provided by a model, not aware of the implementation details of the controller and the business service layer. View components in MVC fulfil key responsibilities as follows:

- A view is responsible for generating Web content from data models created by the controller (possibly with the help of request related information). The view should display data rather than perform data retrieval or handle data retrieval failures.

- A view may need to perform display logic, as distinct from control logic or business logic but unique to a particular presentation.

- Switching between views accessing the same data model should not incur modification to either model or controller code, which is the key to separating presentation from workflow logic.

- Switching between views implemented by different view technology should not incur modification to either model or controller code.

The benefits of specifying view responsibilities in such a limited way lie in maintainability and testability, especially for large Web applications. Both maintainability and testability are of significant importance to software development and reengineering.

5.3.3 Evolvable Web Application Framework

An evolvable WAF must provide a clean and thin Web tier. A clean Web tier could be achieved by a well designed WAF, while a thin Web tier needs the support of the whole infrastructure.

- A WAF should merely make it easy to process user input and display the results, and offer easy interface-based integration with business objects.

- A WAF should have most of its configuration done within the overall application registry.

With the infrastructure introduced in Section 5.2, WAF components can be transparently configured in exactly the same way as application objects, simplifying both the implementation and the use of a framework. A WAF based on the proposed infrastructure should meet the following requirements:

- The WAF should be closely integrated with the proposed infrastructure.
- WAF objects should allow for easy and consistent configuration.
- Application objects should be configurable in the same way by the application registry.
- The framework should explicitly separate the roles of controller, model, and view.
- Application code should have as little dependence as possible on the container API.
- The framework should be extensible via interfaces.
- The framework should completely decouple controllers from views, achieving complete view substitutability.
- The framework should provide a simple MVC implementation with the ability to customise the workflow.

Such a WAF, combined with the proposed infrastructure, is the target framework of both software development and reengineering.

The proposed MVC WAF is implemented in Java Servlet, where the controller is a Servlet and the request controller (sub-controller) a POJO (Plain Old Java Object) object implementing a specific interface.

- The controller Servlet delegates an HTTP request to an application request controller chosen by matching the request against a list of implementations of the ControllerMapping interface.

- The controller Servlet invokes the processRequest() method of request controller to get an object of type ModelAndView representing the name of a view and model data to display.

- The controller Servlet invokes an object of type ViewLocator to obtain a reference to the View object via the name returned by the request controller.

- The controller Servlet invokes the processView() method of the view to generate content.

5.3.3.1 Controller Servlet

In J2EE environment, the entry point of an MVC web application with any framework is a generic controller servlet. In the proposed framework, the controller is a servlet of class ControllerServlet that is responsible for delegating a request to an application-specific request controller for processing.

- To modularise a Web application, multiple controller Servlets can be used, which are associated with their own WebApplicationRegistry extending the ApplicationRegistry of the infrastructure.

- All controller Servlets share a common parent application registry for sharing business objects.

5.3.3.2 Request to Controller Mapping

- The controller servlet chooses which request controller to delegate incoming requests via implementations of the ControllerMapping interface defined in the current bean factory, or matching the request URL to a controller bean name if ControllerMapping implementation is not available.

- Multiple mappings can be defined for each controller servlet. The controller servlet will search all beans in its application registry that implement the ControllerMapping interface, and apply them in a fixed order until one finds a match.

5.3.3.3 Request Controller

- A request controller is essentially an extension of the controller servlet's functionality. By delegating to one of a number of controller objects, the controller servlet remains generic.

- Request controllers implement the Controller interface, which contains a single method processRequest() that returns ModelAndView object containing a data model and the name of a view that can render the model.

- Controllers are multithreaded components, where any instance data should normally be read-only to avoid corrupted states or the need to synchronise access.

- Controllers are JavaBeans that enables their properties to be set in the application registry definition associated with the relevant servlet.

5.3.3.4 Model and View

A controller returns both a model and a view name represented by an object of the ModelAndView class containing model data and the string name of the view that should render the model.

- A model consists of one or more Java objects (usually, JavaBeans). Each model object has an associated name and the complete model is returned as a Map.

- A view is an object for rendering a model. The purpose of the View interface is to decouple controller code from view technology by hiding view implementation behind a standard interface.

- A view does not perform any request processing or initiate any business logic.

A view must implement the View interface, where the most important method is processView that writes output to the response object according to model data. A view may use any one of a number of strategies to implement this method. For example:

- Forwarding to a resource such as a JSP.
- Performing an XSL transformation.
- Using a custom output generation library to generate binary format.

5.3.3.5 ViewResolver and ContextLoaderServlet

The ModelAndView objects returned by controllers contain view names rather than view references, which decouples controller from view technology. View names are resolved via an implementation of the ViewResolver interface stored in a controller Servlet application context.

To integrate the WAF to overall application infrastructure, a root WebApplicationRegistry object must be attached to the ServletContext before any controller servlet works. The root WebApplicationRegistry is created and set as a ServletContext attribute by the ContextLoaderServlet, which must be set to load on startup before any other servlet via the <load-on-startup> web.xml element.

Code examples for applications implemented with different controllers are given in Appendix C. A detailed case study for using the proposed Web Application Framework is given in Chapter 11.

5.4 Summary

The infrastructure described in this chapter enforces programming code to interfaces instead of concrete classes, by using a central configuration manager or application registry. This decouples configuration data and plumbing code from application objects, which only need to expose JavaBean properties and do not have dependencies on the supporting infrastructure.

MVC architectural pattern is still, from the analysis of status quo (Section 3.4), challenges (Section 1.2.4) and problems (Section 5.3.1) of web applications, the best solution to successful web interfaces, although current implementations need many improvements, which are discussed in this Chapter.

MVC theory and the concepts (Section 5.3) are shared between some successful implementations of the MVC pattern (Section 8.2.1.1). The design and use of a simple but powerful web application framework integrated with the infrastructure (Section 5.2) is discussed in this Chapter. The proposed framework is superior to existing ones for several reasons:

- Clear and clean separation between web-tier components and business objects independent of the web interface, which is often not addressed in MVC applications with web-specific code intertwined into business logic.

- No existing MVC web framework satisfies all the proposed design goals.

- This framework is integrated with a unified solution not only to software development but a round-trip engineering process (Section 4.1).

- The proposed framework decouples view technologies from controllers, enabling alternative view implementations.

A case study using this proposed Web Application Infrastructure and Framework is given in Chapter 11.

CHAPTER 6

A Formal Method for High-level Specifications Extraction

A formal method for high level specifications extraction relies on the concept and description of abstraction, which is an effective way to reduce the complexity of software systems, as it is in the case of operating systems for abstraction of underlying various hardware and middleware systems for abstraction of underlying heterogeneous computer systems.

In respect of the purposes of applying abstractions during reverse engineering, all abstraction rules are classified into three categories: elementary abstraction rules, architecture abstraction rules and transformation rules:

- Elementary Abstraction Rules. These abstraction rules eliminate implementation details by recovering lower level specifications such as program logic and algorithm. They focus on a tiny portion of a system from a fine-grained perspective.

- Architecture Abstraction Rules. The purpose of architecture abstraction rules is to simplify relationships. Interconnections between procedures, modules/objects and components that are judged as irrelevant by architecture abstraction rules will be left out of the model representation.

- Transformation Rules. The purpose of transformation rules is to eliminate technology details.

From the technical point of view, the only difference between transformation and abstraction rules is that the former involve two modeling languages with different metamodels, whilst the later, though changing the abstraction level of a processed model, do not produce a model written in a different language. Put in the MDA context, this means that whilst there might be PSMs (Platform Specific Models) of a specific

system at different levels of abstractions (one is less/more concrete than the other), only one PIM (Platform Independent Model) exists for a specific system.

The proposed formal method for high-level specifications extraction is based on the three types of abstraction rules defined above.

6.1 Specification Extraction Framework in MDA

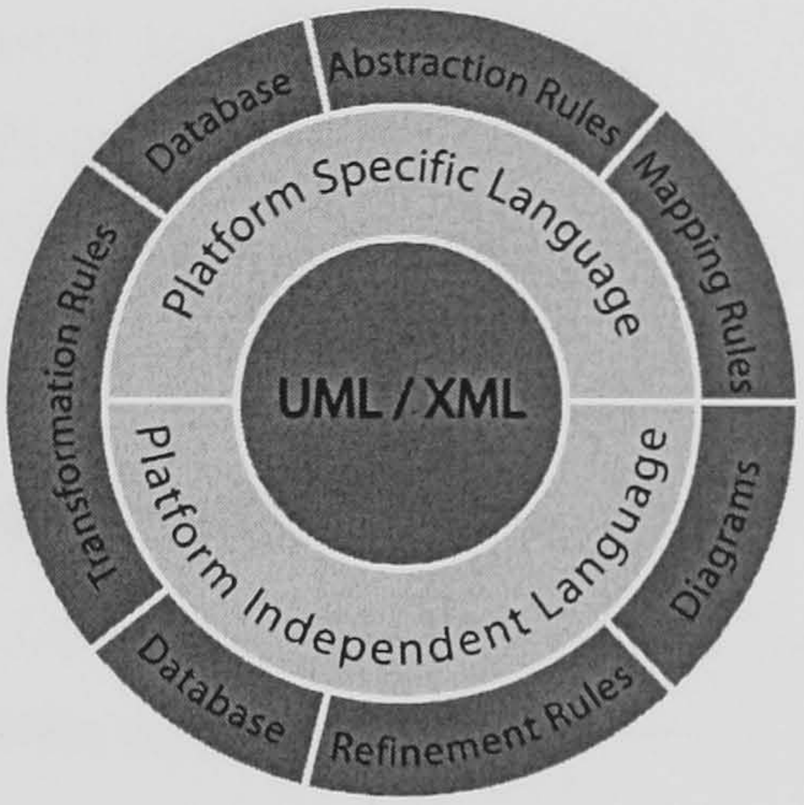

Figure 6-1-1 Specification Extraction Framework

The implementation of the Specification Extraction in MDA is based on three levels of techniques shown in Figure 6-1-1. The very core of this framework is the combination of UML and XML, which provides language support for all the representations, rules and data formats. The middle level is the combination of Platform Independent Language (PIL) and Platform Specific Language (PSL), which define the outer level of the framework. PSL is used to construct translation rules (for translation from original system to PSL code), abstraction rules (for abstractions from initial PSMs to final PSMs) and working databases. PIL is used to construct high level diagrams, refinement rules (for transformations from PIMs to PSMs) and working databases. The construction of transformation rules (for abstractions from PSMs to PIM) rely on both languages.

Translation maps source language to target language in their metamodel level. Abstraction is for recovering higher level specifications from code, which is constructed by the taxonomy of abstraction discussed before. From the technical point of view, there is no significant difference in implementing these abstraction rules, each of which is written in OCL2.0, in accordance to a certain category of abstractions. By defining a

layer of PSL, reverse engineering technique is designed for PSLs, not numerous programming languages. Source code will be translated into one PSL and be tackled by one set of reverse engineering techniques as long as these systems are implemented on a same platform. As the number of platform technologies is much less than that of programming languages, this will significantly reduce the complexity of designing reverse engineering techniques.

In addition to the PSL itself, all kinds of specification extraction rules, including translation, abstraction and transformation rules, are defined in OCL as well. The method used to define specification extraction rules is based on the work of [Kleppe03], which proposes an OCL-based definition of transformation rules

6.2 Requirements for Specification Extraction Rule

A translation rule is used between a programming language and a Platform Specific Language (PSL). The definition of a translation rule should contain the following information:

- Source and target language references.
- Optional translation parameters.
- Source and target language elements from their metamodels.
- The source end invariant stating the conditions that must hold in the source model for this translation rule to apply. The invariant may only be expressed on source language.
- The target end invariant stating the conditions that must hold in the target model for this translation rule to apply. The invariant may only be expressed on target language.
- A set of mapping rules, each of which translates some source model elements to target model elements equivalently.

An abstraction rule is used between two PSL models at different abstraction levels. The definition of an abstraction rule should contain the following information, where the source model has more implementation details than the target model:

- Optional abstraction parameters.

- Source and target language elements from their metamodels.

- The source end invariant stating the conditions that must hold in the source model for this abstraction rule to apply.

- The target end invariant stating the conditions that must hold in the target model for this abstraction rule to apply.

- A set of mapping rules, each of which abstracts some source model elements to target model elements with less implementation details.

A transformation rule is used between a final Platform Specific Model and a PIM. The definition of a transformation rule should contain the following information, where the source model is a final PSM:

- Source and target language references.

- Optional translation parameters.

- Source and target language elements from their metamodels.

- The source language invariant stating the conditions that must hold in the source model for this transformation rule to apply. The invariant may only be expressed on model elements from source language metamodel.

- The target language invariant stating the conditions that must hold in the target model for this transformation rule to apply. The invariant may only be expressed on model elements from target language metamodel.

- A set of mapping rules, each of which transform some source model elements to target model elements without platform specific information.

6.3 Notations for Specification Extraction Rules

There is a set of OCL notations for each of the three types of Specification Extraction rules. Listing 6-3-1 shows the grammar rules for Specification Extraction rules.

```
<ruleType> := "Translation" | "Abstraction" | "Transformation"
<paramsList> := <varName>: <varType> = <expression>;
| <paramsList> <paramsList>
<params> := params <paramsList>
<sourcesList> := <varName>: <modelType>::<modelElement>;
| <sourcesList> <sourcesList>
<sources> := sources <sourcesList>
<targetsList> := <varName>: <modelType>::<modelElement>;
| <targetsList> <targetsList>
<targets> := targets <targetsList>
<sourceInvariant> := source invariant <booleanExpression>;
<targetInvairiant> := target invariant <booleanExpression>;
<mappingRulesListForTranslation> := <operand> <=> <operand>:<ruleName>;
| <operand> <=> <operand>
| <mappingRulesListForTranslation> <mappingRulesListForTranslation>
<mappingRulesListForAbstraction> :=  <operand> ~> <operand>:<ruleName>;
| <operand> -> <operand>
| <mappingRulesListForAbstraction> <mappingRulesListForAbstraction>
<mappingRulesListForTransformation> := <operand> => <operand>:<ruleName>;
| <operand> => <operand>
| <mappingRulesListForTransformation> <mappingRulesListForTransformation>
<mappingRules> := mappings <mappingRulesListForTranslation>
| <mappingRulesListForAbstraction>
| <mappingRulesListForTransformation
<specificationExtractionRule> := <ruleType> <ruleName> (<sourceRef>, <targetRef>) {
<params>
<sources>
<targets>
<sourceInvariant>
<targetInvariant>
<mappingRules>
| <sources>
<targets>
<sourceInvariant>
<targetInvariant>
<mappingRules>
}
```

Listing 6-3-1 Notations for Specification Extraction Rules

Every Specification Extraction rule begins with one of the keywords: "Translation", "Abstraction" or "Transformation" and a name. The source and target languages are

referenced by stating their names between brackets after the rule name. For abstraction rules, the source language is the same with the target one. The rest of the Specification Extraction rule is included between curly brackets. The parameters to Specification Extraction rules are defined as a list of variable declarations following the keyword "params". The source and target language elements are defined as variable declarations following the keywords "source" and "target" respectively. The source and target language invariants are defined as OCL Boolean expressions after the keywords "source invariant" and "target invariant".

All mapping rules come after the keyword "mappings". The symbol <=> is used as a binary operator with two operands, which means that there is a translation rule from the source code to the target code translating the left-hand side operand to the right-hand side operand equivalently. The base of all translation mapping rules is composed of a number of primitive mapping rules, which translate primitive data types in the source language (in this case, Java) to those in the target language (Platform Specific Language).

The symbol ~> is used as a binary operator with two operands, which means that there is an abstraction rule from the source code to the target code abstracting the left-hand side operand to the right-hand side operand at a higher level of abstraction. The base of all abstraction mapping rules is composed of a number of elementary abstraction rules, which abstract the constructs in the source code ("more concrete" Platform Specific Model) to higher level ones in the target code ("more abstract" Platform Specific Model).

The symbol => is used as a binary operator with two operands, which means that there is a transformation rule from the source code to the target code transforming the left-hand side operand to the right-hand side operand with platform specific information left out. The base of all transformation mapping rules is composed of a number of primitive mapping rules, which transform primitive data types in the source language (PSL) to those in the target language (PIL).

6.4 Java/PSL Translation Rules

Ideally, a PSL should be the same language used to write a Specification Extraction rule. This could improve the consistency of modeling and facilitate the implementation of automatic tools. However, OCL itself is not a programming language, but a declarative language, which states what actions to perform, instead of how. A language consisting of programming constructs is still required for PSL. Preferably, this should be a standardised, widely accepted intermediate language, such as CIL (Common Intermediate Language). In this thesis, the programming constructs of Java (while, if, etc.) itself are used as part of PSL. The main purpose is to demonstrate the proposed approach of software reverse engineering based on OCL defined abstraction rules.

An OCL-based translation rule for Java to PSL represents: (1) the translation between Java constructs and PSL defined in OCL; (2) the translation of Java primitives and operators into PSL, or (3) the translation between Java compound statements and PSL.

```
context PSL::Class
def:
  attributes() = feature->select(isTypeOf(Attribute));
  operations() = feature->select(isTypeOf(Operation));

Translation Class2Class (Java, PSL) {
  sources
      c1: Java::Class;
  targets
      c2: PSL::Class;
  source invariant true;
  target invariant true;
  mappings
      c1.attributes()<=> c2.attributes();
      c1.operations()<=> c2.operations();
}
```

Listing 6-4-1 Translation between Java Class and PSL Class

In Listing 6-4-1, a translation rule is defined to translate Java class into PSL class, which denotes that:

- A PSL class consists of attributes and operations.

- Every Java class is translated into a PSL class.

- Every operation in the Java class is translated into an operation in PSL class.

- Every attribute (private, protected, or public) of the Java class is translated into an attribute in the PSL class with the identical visibility.

In this way, PSL is used to define the counterparts of other Java constructs, such as "interface" and "exception", and the corresponding translation rules.

PSL Types	Java Types
in	Integer
float	Real
string	String
boolean	Boolean
Class	OclType
Object	OclAny

Table 6-4-1 Mapping between Java and PSL Types

The translation of Java primitives and operators into PSL is basically a one-to-one mapping. Most of Java datatypes could be directly mapped onto a PSL counterpart that is supported by OCL. In Table 6-4-1, a Listing of such mappings is given. In a similar way, Java operators, such as "+" and "-", can be translated into PSL as well.

```
context PSL::While
def:
  statements() = feature->select(isTypeOf(statement));
  booleanExpression() = feature->select(isTypeOf(booleanExpression));

Translation While2While (Java, PSL) {
  sources
      w1: Java::While;
  targets
      w2: PSL::While;
  source invariant true;
  target invariant true;
  mappings
      w1.statements()<=> w2.statements();
      w1.booleanExpression()<=> w2.booleanExpression();
}
```

Listing 6-4-2 Translation between Java while Statement and PSL while Statement

In Listing 6-4-2, a translation rules is defined to translate a Java compound statement "while loop" into PSL. In a similar way, PSL is used to define the counterparts of other

Java compound statements, such as “for loop” and “if … then”, and the corresponding translation rules.

6.5 Elementary Abstraction Rules

Elementary abstraction rules are given as follows, where the “Entity” is defined for a group of statements in PSL with independent functionality, and "Abstraction" represents the type of all the abstraction rules in PSL.

```
Abstraction Reflexive (PSL, PSL) {
  sources
     p1: PSL::Entity;
  targets
     p1: PSL::Entity;
  source invariant
     true;
  target invariant
     ture;
  mappings
     p1 ~> p1;
}
```

Listing 6-5-1 Reflexive Abstraction Rule

Listing 6-5-1 shows the reflexive abstraction rule, denoting that any entity is the abstraction of itself.

```
Abstraction Transitive (PSL, PSL) {
  params
     a1: PSL::Abstraction;
     a2: PSL::Abstraction;
  sources
     p1: PSL::Entity;
     p2: PSL::Entity;
  targets
     p3: PSL::Entity;
  source invariant
     p1 ~> p2: (a1) and p2 ~> p3: (a2);
  target invariant
     true;
  mappings
     p1 ~> p3: (a1, a2);
}
```

Listing 6-5-2 Transitive Abstraction Rule

Listing 6-5-2 shows the transitive abstraction rule, denoting that a set of abstraction rules can be concatenated to make up a new rule. The source of the new rule is that of the first abstraction rule in the set, and the target of the new rule is that of the last abstraction rule in the set.

```
Abstraction Monotonic (PSL, PSL) {
  params
      a1: PSL::Abstraction;
      a2: PSL::Abstraction;
      f1: PSL::Function;
  sources
      p1: PSL::Entity;
      p3: PSL::Entity;
  targets
      p2: PSL::Entity;
      p4: PSL::Entity;
  source invariant
      p1 ~> p2: (a1) and p3 ~> p4: (a2);
  target invariant
      true;
  mappings
      f1(p1, p3) ~> f1(p2, p4): (a1, a2);
}
```

Listing 6-5-3 Monotonic Abstraction Rule

Any abstraction is actually performed on a part of the whole software system. If the abstractions defined are not monotonic within most common context of software systems, the application of the abstraction rules could be quite limited. Listing 6-5-3 shows the monotonic abstraction rule, where the "Function" represents any functional transformations. The monotonic rules denotes that if entity p1 and p3 can be abstracted by abstraction rules a1 and a2 respectively to produce entity p2 and p4, the result of functional transformation f1, with p1 and p3 as arguments, can be abstracted by the combination of a1 and a2 to the result of functional transformation f1, with p2 and p4 as arguments.

```
Abstraction Weakening (PSL, PSL) {
sources
    p1: PSL::Entity;
    p2: PSL::Entity;
targets
    p3: PSL::Entity;
source invariant
    p1~>include(p2);
```

```
    (p2) isTypeOf(WeakeningType);
  target invariant
    p1->include(p2 + p3);
  mappings
    p1 ~> p3;
}
```

Listing 6-5-4 Weakening Abstraction Rule

Any semantics weakening without contradicting the healthiness invariant but making a clearer understanding of the system can be defined as weakening abstraction rules. Semantics weakening is used to eliminate inessential information, such as exception handling, user interface and detailed comments. Listing 6-5-4 shows the definition of weakening abstraction rule, where entity p1 contains entity p2 that is one of Weakening types, and can be eliminated. The remainder of p1 minus p2 is the weakening abstraction of p1.

```
Abstraction Attribute (PSL, PSL) {
  sources
    c1; PSL::Class;
    a1: PSL::Attribute;
    o1: PSL::Operation;
    o2: PSL::Operation;
    f1: PSL::Function::SetterGetter;
  targets
    c1: PSL::Class;
  source invariant
    o1 ~> f1(a1) or isEmpty(o1);
    o2 ~> return a1;
  target invariant
    true;
  mappings
    a1 = a1;
    a1.addFeature(readonly = (isEmpty(o1)));
    c1 = c1 - o1 - o2;
}
```

Listing 6-5-5 Attribute Abstraction Rule

Java "set" and "get" operations will be eliminated and their corresponding attribute will be abstracted into PSL attribute with appropriate visibility. If the "set" operation is absent, the abstracted PSL attribute will be read-only. Listing 6-5-5 shows the definition of attribute abstraction rule, where class c1 is abstracted. If operations of c1 can be abstracted into semantics of setting or getting attribute a1, the "set" and "get" operations

of c1 will be eliminated, with the addition of a new feature "readonly" for a1, which has a Boolean value determined by the presence or absence of "set" operation o1.

```
Abstraction SequenceFolding (PSL, PSL) {
  sources
     p1: PSL::Entity;
     p2: PSL::Entity;
  targets
     p3:PSL::Entity;
  source invariant
     p1~>p2;
  target invariant
     (p1; p2)~>p3;
  mappings
     p3 = p1 /\ p3;
}
```

Listing 6-5-6 Sequence Folding Abstraction Rule

If no contradiction is caused when substituting the sequential relation between two entities to conjunction relation, then the sequence of entities can be folded through conjunction. In Figure 6-5-6, if two entities p1 and p2 have sequential relationship and the semantics of them imply that of the conjunction of them, the sequential relationship can be replaced by conjunction relationship.

This rule can be applied when the execution order of a sequence is not crucial, which in non-parallel systems, is true under most situations, except any operation provides parts of the pre-conditions of its successor within the sequence. However, in parallel systems, if the sequence relates with communication or shared resources, it can not be folded with conjunction.

The purpose of architecture abstraction rules is to simplify relationships. Interconnections between procedures, modules/objects and components that are judged as irrelevant by architecture abstraction rules will be left out of the model representation. Architecture abstraction rules are applied in each phase of the Specification Extraction process, and executed against different elements. On the other hand, they have one thing in common, for all of them deal with interconnections. Table 6-6-1 lists the architecture related architecture abstraction rules belonging to the abstract process model. They represent basic principles for architecture recovery and need to be instantiated in terms of specific technology domain. Architecture abstraction rules used by modules and

components identification are defined in PSL in a similar way with those of elementary abstraction rules.

6.6 Architecture Abstraction Rules

The translation rules and abstraction rules presented in this thesis are far from enough for reverse engineering a real world Java system. However, the main purpose of this research is to propose an approach to rule definition. Table 6-6-1 shows the architecture abstraction rules.

Rule Name	Definition
Primitive Component	Initial component is defined as primitive component, which can be source file, program module and subsystem.
Component Composition	Component composition is to combine two components when they have direct connectors, such as procedure call or shared variable, may be combined into one component.
Trivial Component	If a part of the system's functionality is considered too "trivial" to be kept in high level abstraction, the components related to this part of functionality are identified as "trivial components", which should be abstracted away in architecture abstraction.
Primitive Connector	Initial connector is defined as direct connector which can be shared resources, function or procedure call, inter-process call, etc.
Connector Composition	Connector composition is to combine two connectors connecting the same component set into one connector.
Trivial Connector	If all the components connecting to a connector are trivial components, this connector is considered trivial connector and should be abstracted away in architecture abstraction.

Table 6-6-1 Architecture Abstraction Rules

6.7 Java-based Compiler for Specification Extraction Rules

The grammar defined for PSL is implemented as a set of Java classes, each of which corresponds to a specific grammar rule. This is actually an Interpreter pattern

[Gamma95], which takes a set of grammar rules of a language to define a class hierarchy representing statements of that language. Symbols on the right-hand side of a grammar rule are instance variables of the classes representing the left-hand side of each rule. In the case of PSL, this simplified grammar would be represented by nine classes: AbstractConstruct, NonTerminalNode, TerminalNode, AssignmentStatement, WhileStatement, DelayStatement, IfElseStatement, Expression and Variable, the first three of which are abstract classes.

```
<statement> := <assignment statement>
             | <while statement>
             | <delay statement>
             | <if then else statement>
             | <statement> <statement>
<assignment statement> := <variable> = <expression>;
<while statement> := while ( <expression> ) { <statement> }
<delay statement> := delay <variable>;
<if else statement> := if ( <expression> ) { <statement> } else { <statement> }
<expression> := <variable>
             | <expression> + <expression>
             | <expression> - <expression>
             | <expression> * <expression>
             | <expression> / <expression>
<variable> := a | b | c | ... | z
```

Listing 6-7-1 Grammar Rules for PSL

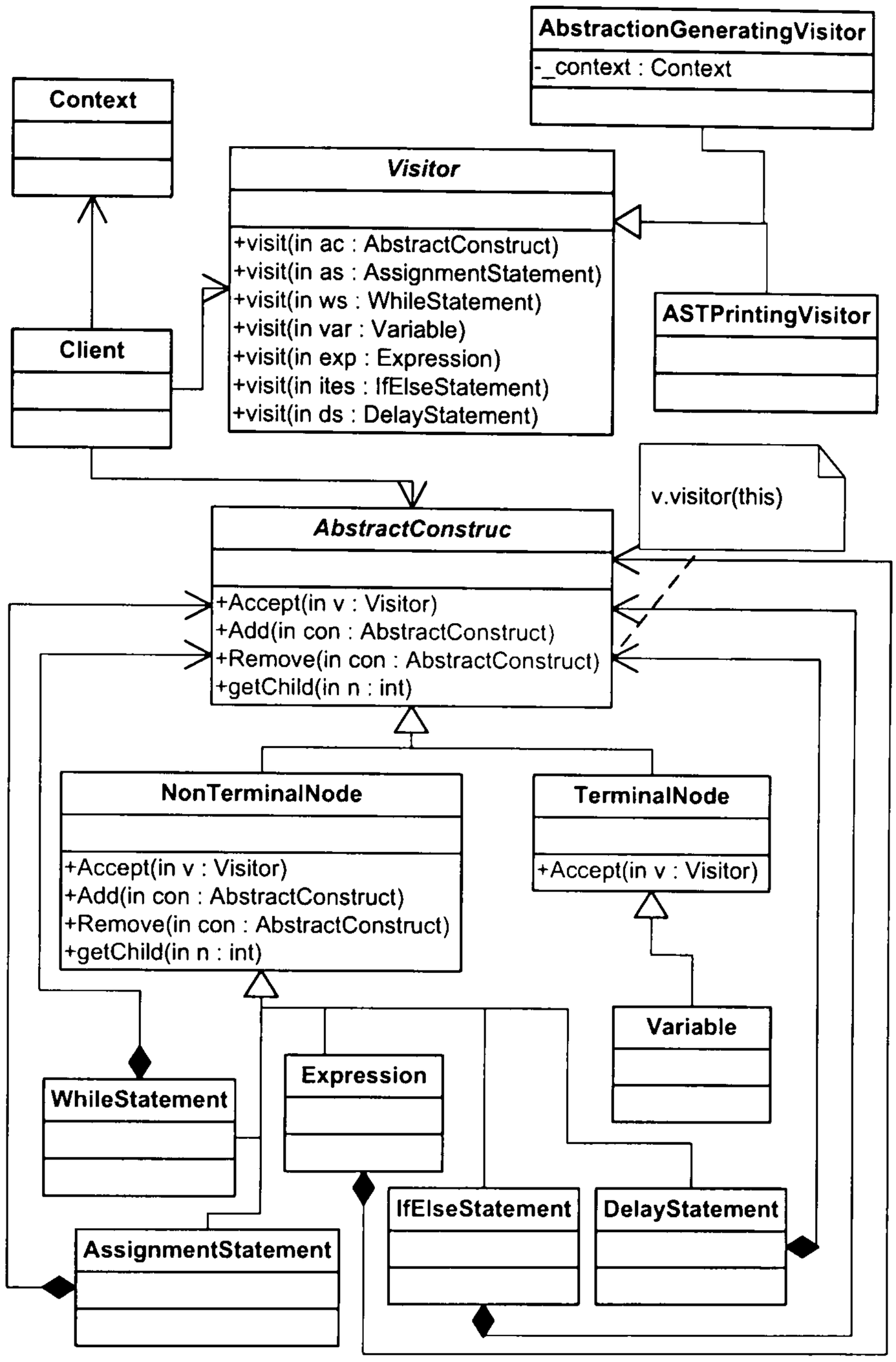

Figure 6-7-1 Simplified Compiler for PSL

Listing 6-7-1 shows the simplified grammar of PSL and Figure 6-7-1 shows the Java implementation of PSL compiler.

In respect of Interpreter Pattern, an abstract Interpret operation needs to be declared in the root class AbstractConstruct and implemented by both terminal and non-terminal subclasses respectively. To interpret a program, a client builds an Abstract Syntax Tree of terminal and non-terminal instances. Then it invokes the Interpret operation passing a

Context object as parameter, which contains global information to the interpreter. The Interpreter pattern is not limited to evaluating an expression. Other "Interpretations", such as pretty printing or type-checking, can be supported by adding new Interpret operations, which means that different Interpret operations exist in a node object. Whilst adding new operations in this way gives rise to changes of all the classes in the Abstract Syntax Tree, using Visitor pattern could obviate this need.

Visitor pattern is useful when a polymorphic operation cannot be included in the class hierarchy for such reasons as that the operation is not designed until finishing the class hierarchy or it would degrade the interface of the classes. In the PSL class hierarchy of nodes, every AbstractConstruct can contain zero or more sub-AbstractConstruct as children. They make up a Composite pattern where objects are composed into tree structures to represent part-whole hierarchies and clients are able to treat individual objects and compositions of objects uniformly [Gamma95]. In this complex hierarchy, distributing all the interpretation-related operations across a class hierarchy results in a system that is hard to implement, change and extend.

Visitor pattern encapsulates related operations from each class in a separate object, and passes it to nodes of the Abstract Syntax Tree as it is traversed. When a node object "accepts" the Visitor, the accept method calls the visit method of the Visitor, passing itself as an argument. The Visitor object receives a reference to each of the instances and can then call its public methods to access data, check types, evaluate expression or generate abstraction code. The implementation of an elementary abstraction rule, Sequence Folding, is given as follows.

```
public abstract class AbstractConstruct {
  protected String name;
  public AbstractConstruct() {
  }
  public void accept(Visitor v) {
    v.visit(this);
  }
  public void add(AbstractConstruct node);
  public void remove(AbstractConstruct node);
  public AbstractConstruct getChild(int n) {
    return null;
  }
}
```

Listing 6-7-2 Class AbstractConstruct

AbstractConstruct, shown in Listing 6-7-2, defines the abstract base class for all classes that define a PSL statement.

```
public class Context {
  public Context() { }
  public boolean applicabilityTest(AbstractConstruct ac1, AbstractConstruct ac2) {
    boolean test = false;
    ...
    return test;
  }
}
```

Listing 6-7-3 Class Context

The class Context, shown in Listing 6-7-3, defines an applicabilityTest method to test if no contradiction is caused when substituting the sequential composition between two representations to conjunction composition.

```
public class NonTerminalNode extends AbstractConstruct {
  Vector children;
  public NonTerminalNode() {
    children = new Vector();
  }
  public void add(AbstractConstruct node) {
    children.add(node);
  }
  public void remove(AbstractConstruct node) {
    children.remove(node);
  }
  public AbstractConstruct getChild(int n) {
    return (AbstractConstruct)children.elementAt(n);
  }
}
```

Listing 6-7-4 Class NonTerminalNode

A NonTerminalNode, shown in Listing 6-7-4, represents a statement such as WhileStatement and AssignmentStatement.

```
public abstract class Visitor {
  Context context;
  public Visitor() {
  }
  public void setContext(Context ct) {
    context = ct;
```

```
  }
  ...
  public abstract void visit(NonTerminalNode var);
  ...
}
```

Listing 6-7-5 Class Visitor

Each abstraction rule is implemented by a specific class that is the subclass of Visitor, shown in Listing 6-7-5. For example, Sequence Folding is implemented by SequenceFoldingVisitor.

```
public class AbstractionGeneratingVisitor extends Visitor {
  public SequenceFoldingVisitor() {
  }
  ...
  public void visit(NonTerminalNode node) {
    Vector children = node.getChildren();
    Iterator i = children.iterator();
    AbstractConstruct ac1 = (AbstractConstruct)i.next();;
    AbstractConstruct ac2;
    for (int index = 0; index < children.size()-1; index++) {
      ac2 = i.hasNext() ? (AbstractConstruct)i.next() : null;
      if (getContext().applicabilityTest(ac1, ac2)) {
        conjunction(ac1, ac2);
      }
      ac1 = ac2;
    }
  } ...
}
```

Listing 6-7-6 Class SequenceFoldingVisitor

The children of a Non-Terminal Node are checked to substitute the sequential composition to conjunction composition, i.e., replacing "semicolon" separators with "logic and" separators. The visit method of SequenceFoldingVisitor Class, shown in Listing 6-7-6, is invoked by the accept method of every non-terminal node that has sub-statements as children (tested by applicabilityTest method of Context Class) and Sequence Folding abstraction rule is applied on every two adjacent non-terminal nodes that can be related by conjunction without giving rise to contradiction.

6.8 Summary

This chapter introduces the concept of abstraction and its relationship with MDA (Model Driven Architecture) and software reengineering. OCL (Object Constraint

Language) is used to specify the syntax of transformation rules that will be applied on MDA models.

CHAPTER 7

A Cognitive Method for Architecture Recovery

One of the major concerns of Web-based systems engineering and reverse engineering is to build or recover various relationships at different abstraction levels and across different domains. For software engineering, techniques, such as nouns identification, Use Cases, CRC cards, can be used to help with identifying entities or objects and analyse their relationships. However, there are no defined processes or diagrams to explicitly and systematically assist in discovering relationships from an implemented software system.

In addition, the existing techniques can not explicitly determine relationships. For Web-based systems, one of key design issues is identifying relationships and implementing them as a variety of links [Fielding98]. Most researches in hypermedia design methodologies claim links for relationships be explicitly modeled as "first class objects" with a set of rich attributes [Christodoulou98, Isakowitz95, Koufaris98, Lange94 and Schwabe96]. However, they do not address the heterogeneous issue of relationships/links, which in Web-based systems include not only URLs linking Web pages, but any relationships defined implicitly for concepts such as configuration, invocation and inheritance.

There are various relationships connecting various domain entities. They are reflected in the implemented software systems. Relationship analysis in both domain space and implementation space can facilitate a round-trip engineering. It helps systems analysts determine the relationship structure of an application and discover potential relationships in application domains. It enhances the understanding of application domains by emphasising the domain conceptual model. For software reverse engineering, it can be used to thoroughly describe an existing system in terms of entities and their relationships. Relationship analysis plays an important role in identifying and modeling relationships in the evolution of software systems.

Relationships are divided into six categories in [Yoo04]. An adaptation of this classification to UML is shown in Figure 7-1-1. The relationship taxonomy defined in [Yoo04] is not compatible with UML, the standard modeling language, and does not have two important relationships required in software development: association and composition. Unlike classic OO design method, it has aggregation defined as a sub relationship of generalization, and has classification and generalisation as two super relationships to all other relationships. The definitions of those relationships have been adapted into a hierarchy that more complies with UML-based OO terminology.

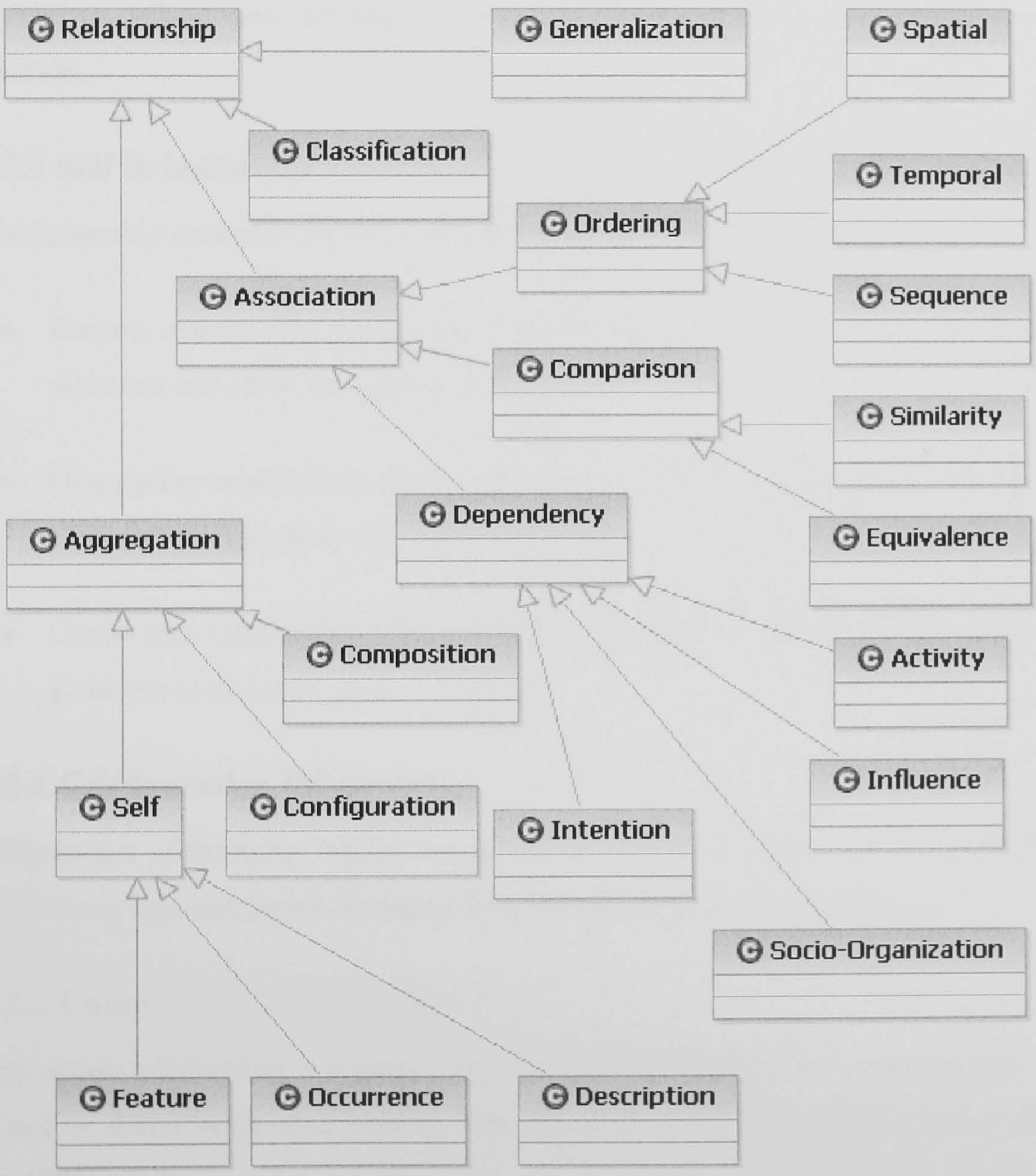

Figure 7-1-1 Hierarchy of Relationships

7.1 Relationship Analysis Principles

7.1.1 Generalisation Relationship

Generalisation is the process of organising the properties of a set of objects that share the same purpose. Generalisation relationship relates an object to the others which have the concepts in a same taxonomy.

7.1.2 Aggregation Relationship

Aggregation is a way of taking a group of distinct objects (parts) to form a whole. Aggregation relationship connects an object of a collection to others or a whole collection.

7.1.2.1 Self Relationship

Self relationship describes the characteristics and distribution of an object.

- Feature relationship relates an object of interest to its attributes, parameters, metadata and other background information.

- Description relationship relates an object of interest to definitions, illustrations, explanations and other descriptive information.

- Occurrence relationship relates multiple instances/views/uses/transformations of an object in different parts of a system.

7.1.2.2 Configuration Relationship

Configuration relationship relates functional or structural settings for a collection of objects to set up a certain environment for executing a process or application.

7.1.2.3 Composition Relationship

Composition relationship is a special aggregation relationship. With composition, parts can not be shared with other objects. The life of the part is completely within the life span of the whole.

7.1.3 Classification Relationship

Classification relationship relates an object of interest to its instance or class.

7.1.4 Association Relationship

An association is a relationship between two objects. It can have three forms including ordering relationship, comparison relationship and dependence relationship.

7.1.4.1 Ordering Relationship

Ordering relationship relates objects with a certain pattern.

- Sequence relationship relates objects in a sequential order.
- Temporal relationship relates objects in a temporal order.
- Spatial relationship relates objects in spatial dimensions.

7.1.4.2 Comparison Relationship

Comparison relationship relates objects by comparing their attributes.

- Equivalence relationship relates instances of the same object to a given one.
- Similarity relationship relates all objects that share similar attributes that could be represented by other relationships.

7.1.4.3 Dependence Relationship

Dependency relationship represents a client-supplier connection in which a change to the supplier requires a change to the client.

- Activity relationship relates objects that are involved in some kind of activity such as input, tools and output.
- Influence relationship relates an object of interest to the object over which it has some kind of influence or control.

- Intentional relationship relates an object of interest to the goals, arguments, issues, decisions, opinions and comments associated with the object. This is more relevant to domain design issue.

- Socio-organisation relationship relates an object of interest to the position, authority, alliance, role and communication associated with the object in a social setting or organisational structure. This is more relevant to domain design issue.

Relationships can be analysed using data mining techniques. For Web-based systems, the number of relationships identified could be enormous. Clustering the relationships to produce meaningful categories can reduce the complexities of these relationships during software reverse engineering. Clustering for Web-based systems often runs after an initial system partition, which produces a number of subsystems for clustering. The initial partition can be performed by the following criteria:

- Relationships that belong to the same stakeholder can be grouped as a single cluster, stakeholders are regarded the topmost element in a whole clustering hierarchy.

- Any important object of the system being analysed can become the topmost element in a cluster hierarchy if it is essential for the problem domain understanding.

- The topmost element of each whole-part hierarchy can be used as the central point for a cluster, where the whole-part relationship can help model complex concepts.

- The topmost element of each generalisation hierarchy can be used as the central point for a cluster, where the generalisation relationship can help model simple but similar concepts.

- Generic relationships can become the criteria for clustering. For example, occurrence relationships can be used for clustering the results according to specific instances of the same element, where the occurrence relationship can

help model systems with heavy use of multiple descriptions of the same element, such as multiple versions of the same design or multiple views of the same scenario.

The above strategies can be combined to have appropriate partitions of the relationship analysis results and should be adapted to chosen clustering techniques. The rest of this chapter focuses on identifying and processing relationships in Web-based systems.

7.2 Relationships in Object-Oriented Systems

Relationships in OO systems include inheritance, package, message, declaration, attribute access and casting, etc. Inheritance, package, message, attribute access and declaration are of most importance to understanding an OO system. A multigraph is used to analyse these relationships, where each of them is represented as an edge with a specific type and connecting two objects. Figure 7-2-1 shows the representation of relationships in OO systems. The numbers in this figure mark the weight of the edge that represents a relationship.

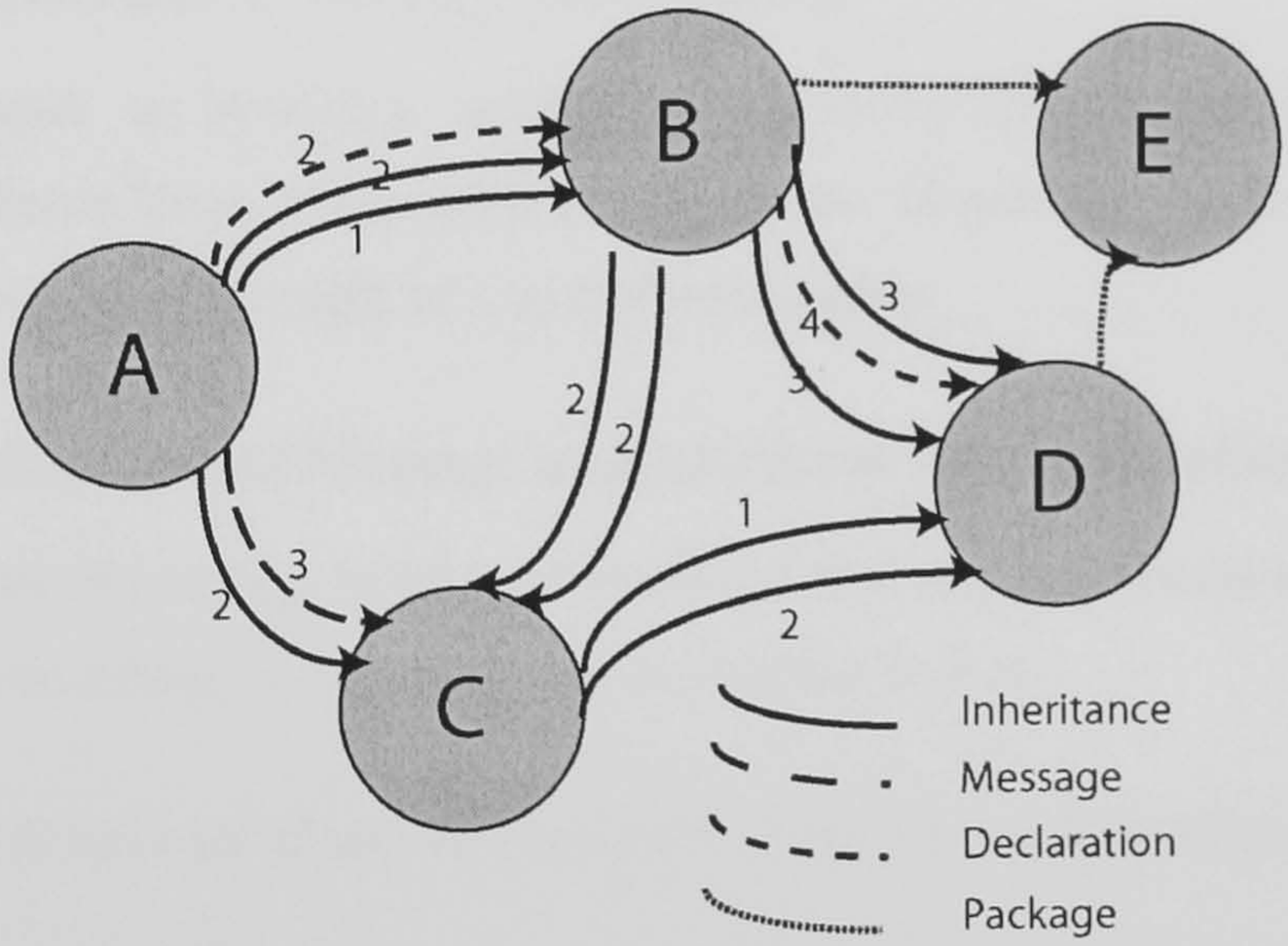

Figure 7-2-1 Representation of Relationships

Relationship edges have attributes defined as follows.

- Source representing the starting node of the edge

- Destination representing the end node of the edge
- Type representing the relationship type
- Weight representing the strength of the relationship

For the purpose of simplicity, a pair of objects can have only one instance of a specific relationship edge, but can have multiple relationship edges for different types.

7.2.1 Quantification of Relationships

The five relationships to be discussed include inheritance relationships, package relationships, attribute access relationships and declaration relationships.

7.2.1.1 Quantification of Inheritance Relationship

Given two classes or interfaces (nodes in the multigraph) A and B, if A extends or implements B then an inheritance edge from A to B exists with a weight of 1.

7.2.1.2 Quantification of Package Relationship

Given two classes or interfaces (nodes in the multigraph) A and B, the package relationship between them is computed by the number of package levels they both share. One package level adds a weight of 1 to this relationship.

7.2.1.3 Quantification of Message and Attribute Access Relationships

The relationships of message (method invocation) and attribute access are conceptually similar and can be treated in the same way as member access.

Some factors will have an impact on the weight of member access relationship.

- Static vs. Instant. Two classes that rely on each other's static members might be considered less coupled than on instant ones.
- Location of Member Access. The location where access occurs might have impact on the weight of relationship. For example, member access in loops can be considered having a stronger coupling than in a non-loop situation.

- Polymorphism. Method overriding makes it difficult to tell which method body a certain invocation links to. Techniques such as Rapid Type Analysis (RTA) [Rayside00] can be used to address this issue.

- Nonlinear Member Accesses combination. The computation of coupling due to member accesses can not be produced by simply summing up the number of accesses, where the degree of coupling may vary for two pairs of classes with the same number of inter-member accesses but different combination of them.

$$\rho(A,B)_{message} = \frac{messages}{senders} + \left(\frac{senders}{|A|}\right)_{significance=2} + \frac{receivers}{100|B|}$$

(7-1)

Formula 7-1 defines the quantification of the message relationship between two classes.

- $\rho(A,B)_{message}$ represents the weight of the message relationship between class A and class B.

- *messages* represents the total number of messages sent to class B from class A.

- *senders* represents the number of methods in A from within which messages are sent to class B.

- *receivers* represents the number of methods in B invoked from class A to send messages.

- $|A|$ and $|B|$ represent the total number of methods contained in classes A and B respectively.

The first factor could be interpreted as the number of average messages sent by the methods of class A to class B that participate in the message sending. The second factor could be interpreted as the proportion of methods in class A that send messages to class B. The last factor is the proportion of methods in class B that are invoked by methods in class A. Note that the second factor has a significance of 2 and the last one is divided by

100. In this way, the weight of the message relationship represents all of the three factors but with a decreasing emphasis on them. For the last two factors, a percentage of 100 will be replaced by two digits of 9.

The quantification of attribute access relationship would be very similar to that of message relationship, as shown in formula 7-2.

$$\rho(A,B)_{attrAccess} = \frac{attrAccesses}{accessors} + \left(\frac{accessors}{|A|}\right)_{significance=2} + \frac{attrsAccessed}{100|B|}$$

(7-2)

7.2.1.4 Quantification of Declaration Relationship

Declaration relationship exists where the name of a class appears within the definition of another. Declarations can be divided into four categories including attribute declaration, instant variable declaration, method parameter declaration and method return type. Given class A and class B, formula 7-3 is used to quantify the declaration relationship between them.

$$\rho(A,B)_{decl} = \frac{paramDecls_A + varDecls_A + rtnDecls_A}{relvntMethods_A} + \left(\frac{attrDecls_A}{|A|_{attr}}\right)_{significance=2} + \frac{relevantMembers_A}{100|A|_{member}}$$

(7-3)

- $attrDecls_A$ is the number of attributes in class A that have the name of class B in their declarations.

- $paramDecls_A$ is the number of method parameters in class A that have the name of class B in their declarations.

- $varDecls_A$ is the number of instant variables in the methods of class A that have the name of class B in their declarations.

- $rtnDecls_A$ is the number of method return types in class A that have the name of class B in their declarations.

- $relvntMethods_A$ is the number of involved methods in class A.

- $relvntMembers_A$ is the number of involved members (attributes and methods) in class A.

- $|A|_{member}$ is the total number of members (attributes and methods) contained in classes A and B respectively.

- $|A|_{attr}$ is the number of attributes contained in class A.

The first factor could be interpreted as the number of average declarations involving class B for all the relevant methods of class A that have the name of class B in their declarations. The second factor could be interpreted as the proportion of declarations involving class B for all the attributes of class A. The last factor is the proportion of members (attributes and methods) in class A that have the name of class B in their declarations to all the members in class A. Note that the second factor has a significance of 2 and the last one is divided by 100. In this way, the weight of the message relationship represents all of the three factors but with a decreasing emphasis on them. For the last two factors, a percentage of 100 will be replaced by two digits of 9.

7.2.2 Unification of Relationships

The unification of relationships will produce a digraph representation of the relationships from the multigraph structure with a set of parameters in this stage. The digraph is a representation suitable for clustering by unifying different types of relationships between classes into a single measurable relationship. Given class A and class B, formula 7-4 is used to quantify the unified relationship between them.

$$\rho_{unification}(A,B) = \sum_{i=1}^{N} \rho(A,B)_i \times w_i \qquad (7\text{-}4)$$

- N is the number of different types of relationships (Inheritance, Package, Message, Attribute Access and Declaration) between class A and class B.

- $\rho(A,B)_i$ is the quantification of the edge with relationship i and connecting class A and class B.

- w_i is a parameter representing the weight associated to the i^{th} term in this weighted sum.

The set of parameters w_i give relative weight of the different types of relationships in the unification of relationships. The weight values for specific relationships need to be decided by domain experts or analysed using other techniques such data mining and further experiments are needed to the appropriate values under certain circumstances. In this case, a general observation is that a single message or declaration relationship is less weighted than an inheritance relationship.

7.2.3 Clustering of Relationships

The clustering of relationships aims to help identify the module structure or architecture of the software system as mentioned in Chapter 4. Different clustering algorithms may be applied in this stage and the resulting clusters represent candidates for future modules, components and subsystems.

7.2.4 Hierarchical Clustering

Hierarchical clustering algorithms produce hierarchies instead of flat clusters. Agglomerative Hierarchical Methods produce dendrograms, which show hierarchical relations between objects in form of a tree of equal branch lengths. Any agglomerative hierarchical classification can be fully described by means of a cophenetic matrix, where a cophenetic similarity (or distance) of two objects is defined as the similarity (or distance) level at which those objects become members of the same cluster. Any dendrogram can be uniquely represented by a matrix in which the similarity (or distance) for a pair of objects is their cophenetic similarity (or distance).

Three criteria for determining distance between arbitrary clusters A and B are described as follows [Larose04].

- Single linkage, or the nearest-neighbor approach, is based on the minimum distance between any record in cluster A and any record in cluster B, where cluster similarity is determined by the similarity of the most similar members from each cluster. Single linkage tends to form long, slender clusters, which may sometimes lead to heterogeneous records being clustered together.

- Complete linkage, or the farthest-neighbor approach, is based on the maximum distance between any record in cluster A and any record in cluster B, where cluster similarity is determined by the similarity of the most dissimilar members from each cluster. Complete-linkage tends to form more compact, spherical clusters, with all records in a cluster within a given diameter of all other records.

- Average linkage is designed to reduce the dependence of the cluster-linkage criterion on extreme values, such as the most similar or dissimilar records, where the criterion is the average distance of all the records in cluster A from all the records in cluster B. The resulting clusters tend to have approximately equal within-cluster variability.

To illustrate hierarchical algorithms, an undirected graph structure of the relationships is used to perform an agglomerative hierarchical clustering using the single linkage update rule. Given two clusters A and B, the single linkage update rule is shown in formula 7-5.

$$\delta(U,C) = \max\left(\delta(A,C),\delta(B,C)\right) \text{ , where } \delta(A,B) = \frac{1}{\rho(A,B)}\text{, } U \text{ is union of A and B} \quad (7\text{-}5)$$

Figure 7-2-2 shows the process of applying agglomerative clustering, where classes A, B, C and D are finally clustered into one cluster.

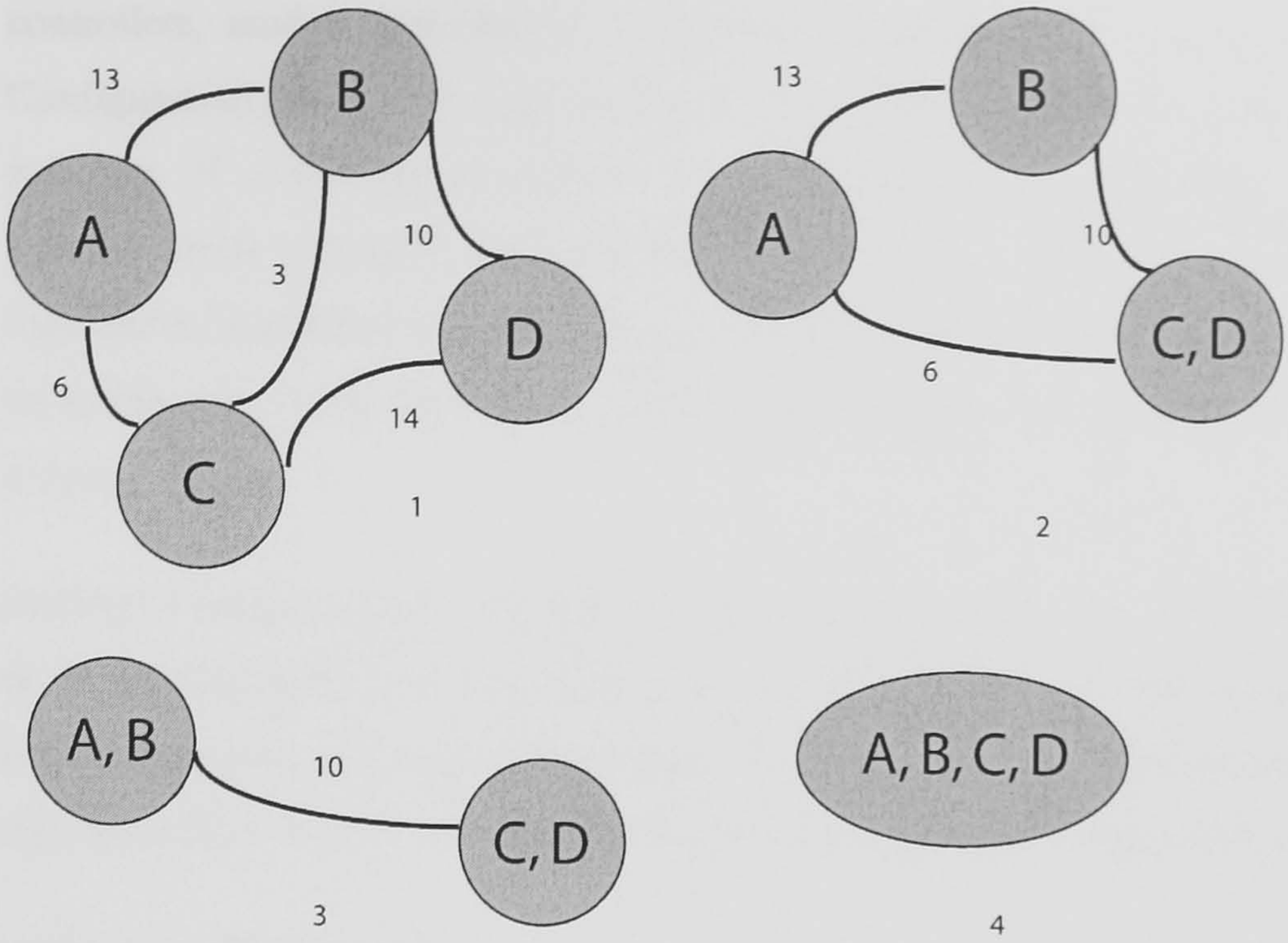

Figure 7-2-2 Process of Agglomerative Clustering

7.3 Relationships in Web-based Systems

Relationships in Web-based Systems vary according to specific architecture. Generally they can be divided into four categories.

- Relationships in OO applications. These are the same relationships discussed earlier including inheritance, package, message, attribute access and declaration.

- Relationships in HTML documents. These are mainly represented by various URL links in Web pages. In addition, relationships can be found in the semantics of the content of the HTML tags in the same or different HTML documents.

- Relationships in Server pages. These relationships include those appearing in HTML documents. In addition, embedded scriptlets may have relationships that are implied by the classes referenced within them.

- Relationships in configuration files. There are often a number of configuration files for a Web-based system, which are used for setting up mappings between

controllers, models and views, and various database or file system accesses. Configuration files are of significant importance for identifying the architectural structure of a Web-based system. For modern Web-based systems such as Apache Struts [struts05] and iBATIS [iBatis05], their configuration files are regarded as documentation of the design, which constrain the developers to have an architectural view during development, and facilitate the understanding of an existing system.

The clustering of relationships in Web-based systems is similar to that in OO systems. However, those relationships are heterogeneous, i.e., derived from different components with different structures and language elements. A set of relationships discovered from the configuration files used in case study of this research is shown in Appendix D.

7.4 Architecture Description

The constituent elements of an application are bound by relationships that define a systematic structure. Understanding such relationships is vital to understand the whole system. During software development, the architecture description of a system represents a set of relationships at a certain level of abstraction and evolves from the blueprint of interacting domain objects to abstract models to data/control flow of program code. Software reverse engineering is carried out in an opposite direction. This section discusses an architecture description language that can be applied on representing a Web-based system at different stages of its evolution process.

In [Kru95], software architecture can be documented via 4+1 views, where the 4 views presented in the paper include logical, process, deployment and data, the '+1' view is the use case view that is a summary of the most significant use cases or scenarios for the system architecture. This model can be extended to N+1 views with the development in modern software systems. A list of common views is shown as follows [Larman04].

- Logic View. Logic view describes conceptual structure and functionality of a software system in respect of layers, subsystems, packages, frameworks, classes and interfaces. It shows major use-case realisation scenarios that illustrate key features of the system.

- Process View. Process view describes the responsibilities, collaborations and the allocation of logical elements of processes and threads.

- Deployment View. Deployment view describes physical deployment of processes and components to processing nodes and the physical network configuration between nodes.

- Data View. Data view describes the overview of the data flows, persistent data schema, schema mapping from objects to persistent data, mechanism of mapping from objects to a database, database stored procedures and triggers.

- Security View. Security view describes the overview of the security schemes and points within the architecture that security is applied, such as HTTP authentication, database authentication.

- Implementation View. Implementation model represents the real materials such as source code, executables, Web pages, DLLs, which are organised into packages or JAR. The implementation view describes the organisation of deliverables and their source, e.g. the source code.

- Development View. Development view describes information required to understand the setup of the development environment.

- Use case View. Use case view describes the use cases significant to architecture and their non-functional requirements.

- This research focuses on the system structure analysis, the outcome of which is represented in a logic view.

To represent logic view of the recovered information, an Architecture Description for Software Reengineering (ADSR) is developed, which represents the relationships between components and connectors in a software system.

ADSR is a language defined as UML profiles to describe software architecture recovered via reverse engineering techniques. It consists of constructs for describing

three main elements of software architecture [Medvidovic2000]: component, interface and configuration.

7.4.1 Architecture Representation in UML

The most important attributes of a component are "interface" and "semantics". In ADSR, the interface of the component is a collection of service ports provided or required, and the semantics of a component can be deduced from the extracted formal specification.

Architecture description for software reengineering could be achieved using UML via UML profiles. A profile is a collection of stereotype definitions, tag definitions and constraints used to represent a specific domain or target. Figure 7-4-1 shows the UML diagram for component description, consisting of three classes, one datatype and one enumeration.

- Interface. ADSR supports specification of component interfaces that consists of interface points, i.e., ports, for participating in a specific connection. In ADSR interface points have one of two properties: provided and required that represent the direction of the connection.

- Type. As said before, ADSR does not have predefined component types. It distinguishes components by their names.

- Semantics. The way to describe semantics of ADSR components can actually be represented in any suitable formal language, such as ITL [ITL].

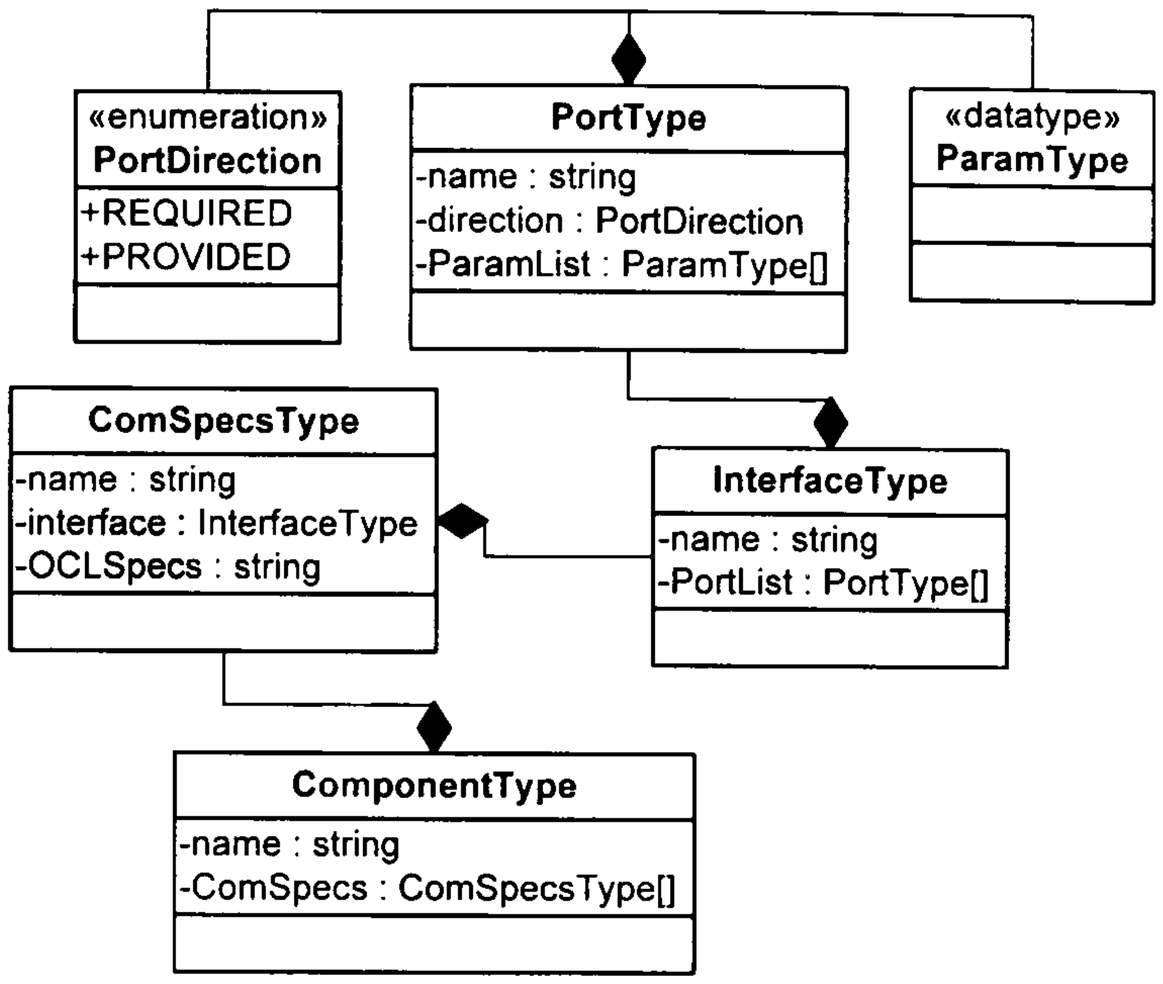

Figure 7-4-1 UML Diagram for Component

Figure 7-4-2 shows the UML diagram for connector description, consisting of 3 classes and one enumeration.

- Interface. The connectors of ADSR have roles as counterparts of components. A connection between a component and a connector is established through the ports of the component and the corresponding roles of the connector.

- Type. As said before, ADSR has predefined connector types that are set by the attribute "name" of class "ConnectorType".

- Semantics. ADSR can use any suitable formal language, such as ITL, to give connectors their semantics.

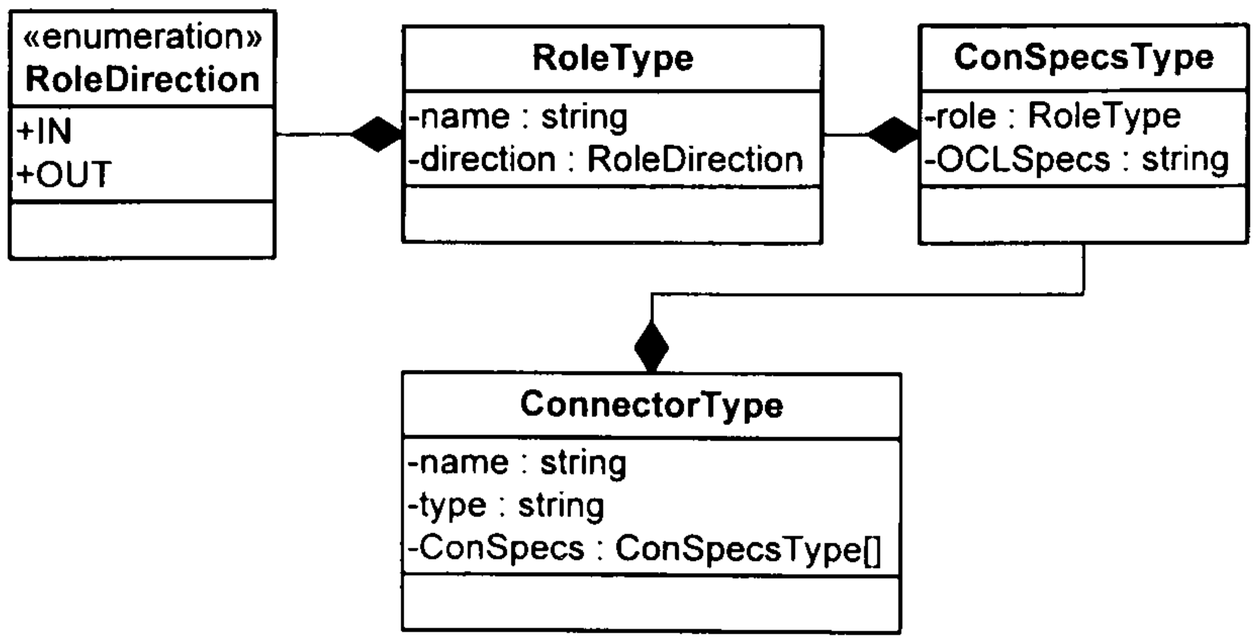

Figure 7-4-2 UML Diagram for Connector

Figure 7-4-3 shows the UML diagram for configuration, consisting of two classes. The semantics of configuration is obtained by analysing the semantics of its connections, which in turn rely on its connectors and components.

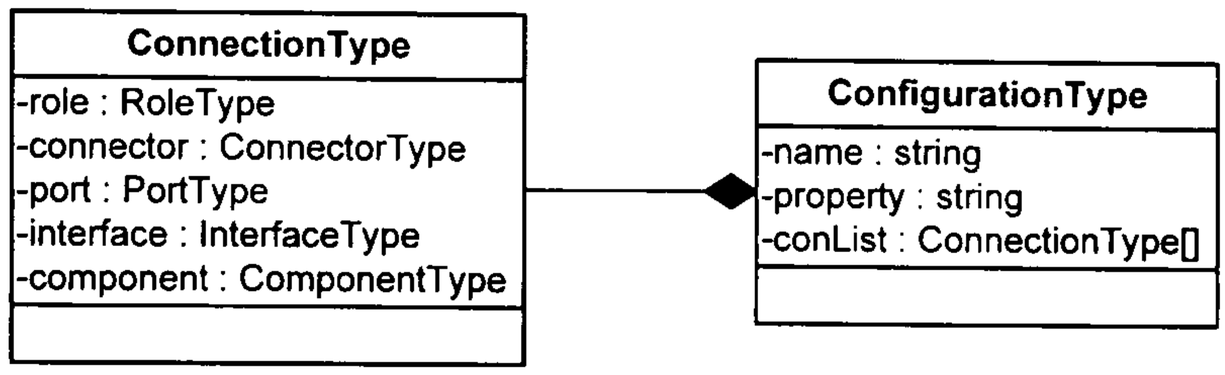

Figure 7-4-3 UML Diagram for Configuration

Figure 7-4-4 shows a complete architecture description using ADSR.

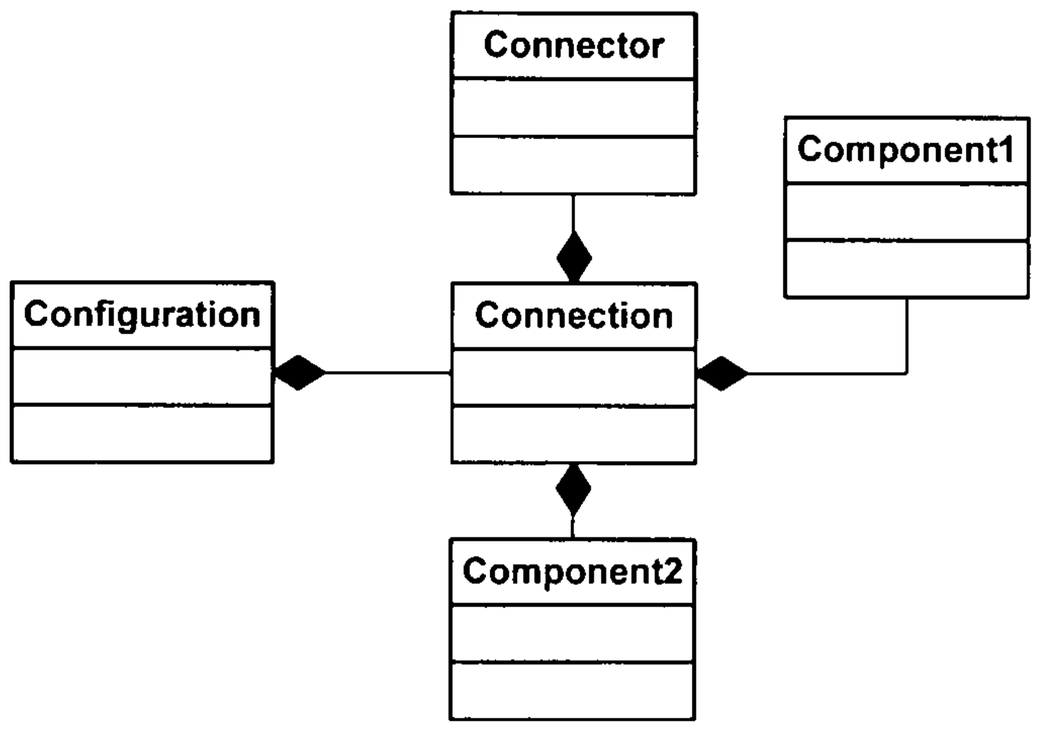

Figure 7-4-4 UML Diagram for Component & Connector

7.4.2 Architecture Description in XML

In Listing 7-4-1, a component description is specified using an XML element "COMPONENT" of type "ComponentType", which consists of an Attribute "NAME" and one or more elements "SPECS", representing the semantics of the component. A "SPECS" is an element of "ComSpecsType", which has two elements "INTERFACE" and "ITLSPECS", and an attribute "NAME".

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
...
```

```
<xsd:simpleType name="PortDirection">
<xsd:restriction base="xsd:string">
<xsd:enumeration value="REQUIRED"/>
<xsd:enumeration value="PROVIDED"/>
</xsd:restriction>
</xsd:simpleType>
<xsd:complexType name="ParamType">
<xsd:sequence>
<xsd:element name="PARAM" type="xsd:string"/>
</xsd:sequence>
</xsd:complexType>
<!-- Definition for PortType that represents
an interface point or player of an interface -->
<xsd:complexType name="PortType">
<xsd:sequence>
<xsd:element name="PARAMETER" type="ParamType"
minOccurs="0" maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="NAME" type="xsd:string"/>
<xsd:attribute name="DIRECTION" type="PortDirection"/>
</xsd:complexType>
<!-- Definition for InterfaceType that represents
interfaces used by components to communicate with
connectors -->
<xsd:complexType name="InterfaceType">
<xsd:sequence>
<xsd:element name="PORT" type="PortType"
minOccurs="1" maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="NAME" type="xsd:string"/>
</xsd:complexType>
<!-- Definition for ComSpecsType that represents
a specification of an interface of a component -->
<xsd:complexType name="ComSpecsType">
<xsd:sequence>
<xsd:element name="INTERFACE" type="InterfaceType"/>
<xsd:element name="ITLSPECS" type="xsd:string"/>
</xsd:sequence>
<xsd:attribute name="NAME" type="xsd:string"/>
</xsd:complexType>
<!-- Definition for ComponentType that
represents a component consisting of interfaces
and specifications -->
<xsd:complexType name="ComponentType">
<xsd:sequence>
<xsd:element name="SPECS" type="ComSpecsType"
minOccurs="1" maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="NAME" type="xsd:string"/>
</xsd:complexType>
<!-- Definition for COMPONENT -->
<xsd:element name="COMPONENT" type="ComponentType"/>
...
</xsd:schema>
```

Listing 7-4-1 XML Schema for Component

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
...
<xsd:simpleType name="RoleDirection">
<xsd:restriction base="xsd:string">
<xsd:enumeration value="IN"/>
<xsd:enumeration value="OUT"/>
</xsd:restriction>
</xsd:simpleType>
<xsd:complexType name="RoleType">
<xsd:attribute name="NAME" type="xsd:string"/>
<xsd:attribute name="DIRECTION" type="xsd:string"/>
</xsd:complexType>
<!-- Definition for ConSpecsType that represents
a specification of a connector-->
<xsd:complexType name="ConSpecsType">
<xsd:sequence>
<xsd:element name="ROLE" type="RoleType"/>
<xsd:element name="ITLSPECS" type="xsd:string"/>
</xsd:sequence>
</xsd:complexType>
<!-- Definition for ConnectorType that represents
the types that a connector can belong to -->
<xsd:complexType name="ConnectorType">
<xsd:sequence>
<xsd:element name="SPECS" type="ConSpecsType"
minOccurs="1" maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="NAME" type="xsd:string"/>
<xsd:attribute name="TYPE" type="xsd:string"/>
</xsd:complexType>
<!-- Definition for CONNECTOR -->
<xsd:element name="CONNECTOR" type="ConnectorType"/>
...
</xsd:schema>
```

Listing 7-4-2 XML Schema for Connector

Connectors intermediate between components and can be used to aggregate components by abstraction rules. A group of roles are used to bind the corresponding service points of components. In ADSR, connector types are classified into two categories according to the context in which a connection occurs.

Local connectors can directly connect different components without relying on network protocols.

- LocalCall represents procedure call (structural system) or method invocation (OO system).

- FileIO represents the file system operations.

- SharedData represents global shared variables.

Distributed connectors implement inter-process communications across networks.

- SocketCnn represents Socket connections, which is the abstraction of UDP and TCP protocols.

- RemoteCall represents RPC (Remote Procedure Call, structural system) or RMI (Remote Method Invocation, OO system) connections.

- GroupCnn represents group connections.

- MsgPassing represents message passing connections.

In Listing 7-4-2, a connector description is specified using an XML element "CONNECTOR" of type "ConnectorType", which consists of two attributes "NAME" and "TYPE" and one or more elements "SPECS", representing the semantics of the connector. A "SPECS" is an element of "ConSpecsType", which has two elements "ROLE" and "ITLSPECS".

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
...
<xsd:complexType name="ConnectionType">
<xsd:attribute name="ROLE" type="xsd:RoleType"/>
<xsd:attribute name="CONNECTOR" type="xsd:ConnectorType"/>
<xsd:attribute name="PORT" type="xsd:PortType"/>
<xsd:attribute name="INTERFACE" type="xsd:InterfaceType"/>
<xsd:attribute name="COMPONENT" type="xsd:ComponentType"/>
</xsd:complexType>
<!-- Definition for ConfigurationType -->
<xsd:complexType name="ConfigurationType">
<xsd:sequence>
<xsd:element name="PROPERTY" type="xsd:string"
minOccurs="0" maxOccurs="unbounded"/>
<xsd:element name="CONNECTION" type="ConnectionType"
```

```
minOccurs="1" maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="NAME" type="xsd:string"/>
</xsd:complexType>
<!-- Definition for CONFIGURATION -->
<xsd:element name="CONFIGURATION" type="ConfigurationType"/>
...
</xsd:schema>
```

Listing 7-4-3 XML Schema for Configuration

The configuration presents interconnection relationship between components and connectors of the (sub-) system. It is the structure description of the whole architecture. In Listing 7-4-3, a configuration description is implemented using an XML element "CONFIGURATION" of type "ConfigurationType", which consists of any number of elements "PROPERTY" and one or more elements "CONNECTION", and an attribute "NAME". A "CONNECTION" is an element of type "ConnectionType", which has five attributes "ROLE", "CONNECTOR", "PORT", "INTERFACE" and "COMPONENT".

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
...
<xsd:complexType name="ModuleType" mixed="true">
<xsd:attribute name="NAME" type="xsd:string"/>
<xsd:attribute name="TYPE" type="xsd:string"/>
<xsd:attribute name="LOCATION" type="xsd:string"/>
</xsd:complexType>

<xsd:complexType name="RelationType" mixed="true">
<xsd:attribute name="NAME" type="xsd:string"/>
<xsd:attribute name="TYPE" type="xsd:string"/>
</xsd:complexType>

<xsd:complexType name="MRRecordType">
<xsd:sequence>
<xsd:element name="MODULE" type="ModuleType"/>
<xsd:element name="MODULE" type="ModuleType"/>
<xsd:element name="RELATION" type="RelationType"/>
<xsd:element name=”BELIEF” type=”integer”/>
</xsd:sequence>
</xsd:complexType>

<xsd:complexType name="MRDatabaseType">
<xsd:sequence maxOccurs="unbounded">
<xsd:element name="RECORD" type="MRRecordType"/>
</xsd:sequence>
</xsd:complexType>

<!-- Definition for MRDatabase -->
```

```
<xsd:element name="MRDATABASE" type="MRDatabaseType"/>
...
</xsd:schema>
```

Listing 7-4-4 XML Schema for M&R Database

Working databases are used to store patterns, relationships and belief values. Nowadays almost all of the major database management systems have either built-in XML support or XML add-ons. Therefore, XML-based database description is the best choice for implementing the working databases required during the reengineering process.

In Listing 7-4-4, the description of M&R database is implemented using an XML element "MRDATABASE" of type "MRDatabaseType", which consists of any number of elements "RECORD" of type "MRRecordType", representing a relationship between two modules with a belief value for evaluating the reliability of this relation.

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
...
<xsd:complexType name="ComponentType" mixed="true">
<xsd:attribute name="NAME" type="xsd:string"/>
<xsd:attribute name="TYPE" type="xsd:string"/>
<xsd:attribute name="LOCATION" type="xsd:string"/>
</xsd:complexType>
<xsd:complexType name="ConnectorType" mixed="true">
<xsd:attribute name="NAME" type="xsd:string"/>
<xsd:attribute name="TYPE" type="xsd:string"/>
</xsd:complexType>
<xsd:complexType name="CCRecordType">
<xsd:sequence>
<xsd:element name="COMPONENT" type="ComponentType"/>
<xsd:element name="COMPONENT" type="ComponentType"/>
<xsd:element name="CONNECTOR" type="ConnectorType"/>
</xsd:sequence>
</xsd:complexType>

<xsd:complexType name="CCDatabaseType">
<xsd:sequence maxOccurs="unbounded">
<xsd:element name="RECORD" type="MRRecordType"/>
</xsd:sequence>
</xsd:complexType>

<!-- Definition for CCDatabase -->
<xsd:element name="CCDATABASE" type="CCDatabaseType"/>
...
</xsd:schema>
```

Listing 7-4-5 XML Schema for C&C Database

In Listing 7-4-5, the description of C&C database is implemented using an XML element 'CCDATABASE' of type 'CCDatabaseType', which consists of any number of elements 'RECORD' of type 'MRRecordType', representing a relationship between two components.

7.5 Summary

This chapter discusses a cognitive way of discovering knowledge in software systems.

- A relationship analysis approach is presented to find relations between various components.

- A clustering algorithm is defined to build architectural view from the discovered relations.

- A UML-based architecture description language is also defined to represent the recovered architecture of a software system.

CHAPTER 8

Patterns for Models Identification in Web-based Systems Evolution

8.1 Domain Architectural Patterns

Patterns are used throughout the evolution process from domain analysis to legacy system migration or integration. They can be classified by abstraction levels into domain architectural patterns, and system architectural and design patterns. A number of system architectural patterns are regarded as the fourth category: integration patterns. They are special for their capabilities to connect heterogeneous systems. This section will introduce domain Architectural patterns. The next two sections will introduce the other two categories.

A domain architectural pattern stands for the abstraction of a set of applications with similar functionality, behaviour and structure [Duffy04]. It describes the essential features in some business domain. By using domain architectural patterns, an assumption is made that any application for a given domain has an abstract description that stands for all its kind. An application belongs to a category if there is an appropriate matching between its attributes and that of the more general architecture. Domain architectural patterns are used in constructing concept model (4.4.1) and architecture migration (4.4.7). They can be divided into five categories.

8.1.1 Management Information Systems

This process is to produce decision-support information based on low-level or transaction input data from various sources. The main activities taken in this process are:

- Register, validate and create basic transaction objects
- Consolidate and aggregate transaction objects
- Present, dispatch and report on consolidated data.

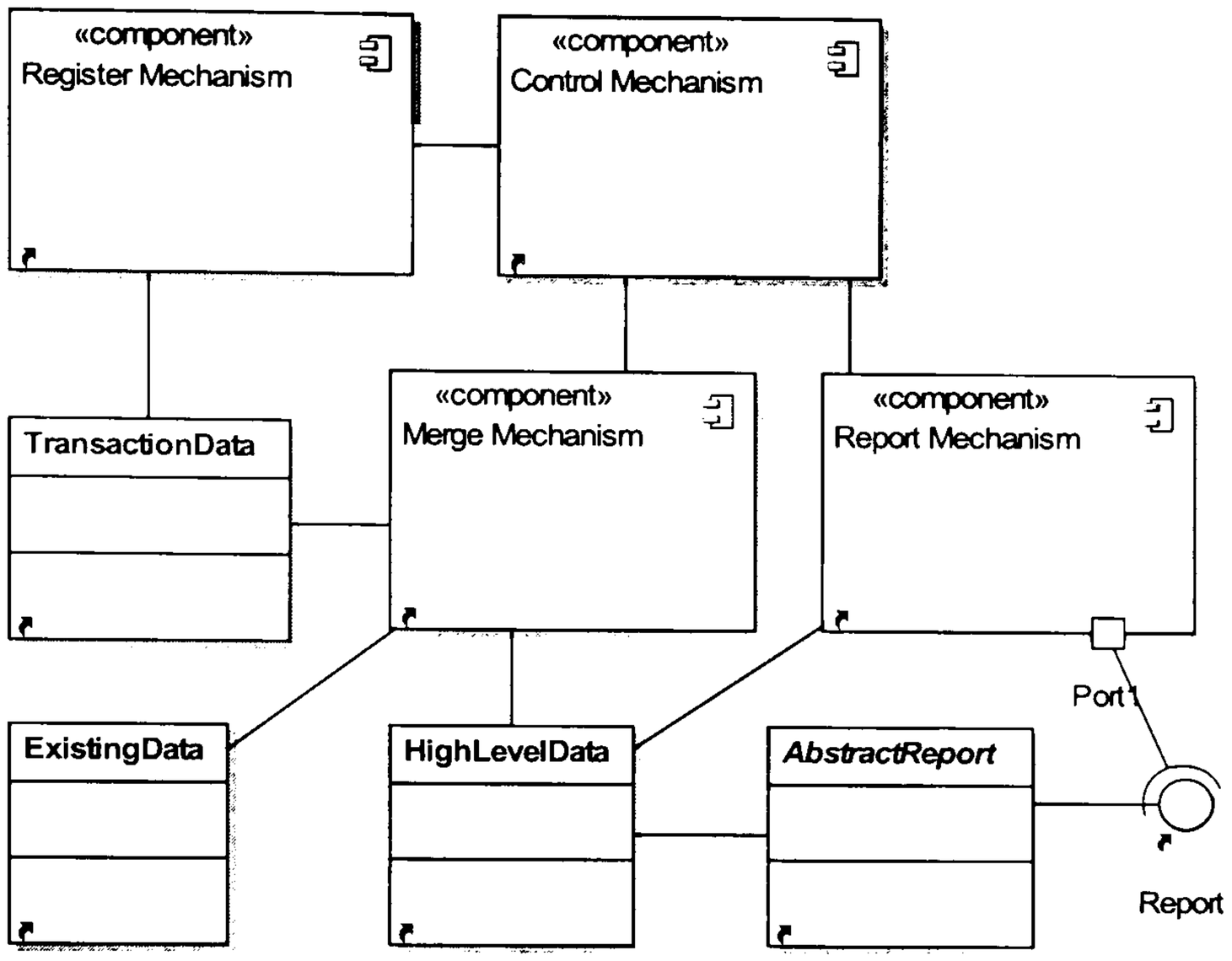

Figure 8-1-1 MIS Domain Pattern

In general, a management information system (MIS) consists of the following components.

- Control Mechanism. This is the control centre that manages the databases such as permanent database, transaction database, data source, and other components.

- Register Mechanism. It reads from source data to transaction database and processes transaction data.

- Merge Mechanism. This component is responsible for merging transaction records with master records. It reads transaction data and existing data (permanent database) to produce high level data.

- Report Mechanism. This component uses high level data produced by merge mechanism to produce various report.

The MIS domain pattern is shown in Figure 8-1-1.

8.1.2 Resource Allocation and Tracking systems

The core process produces status information and the main activities are:

- Register and verify the request
- Assign resources to execute the request
- Monitor the status of the request and present this to stakeholders.

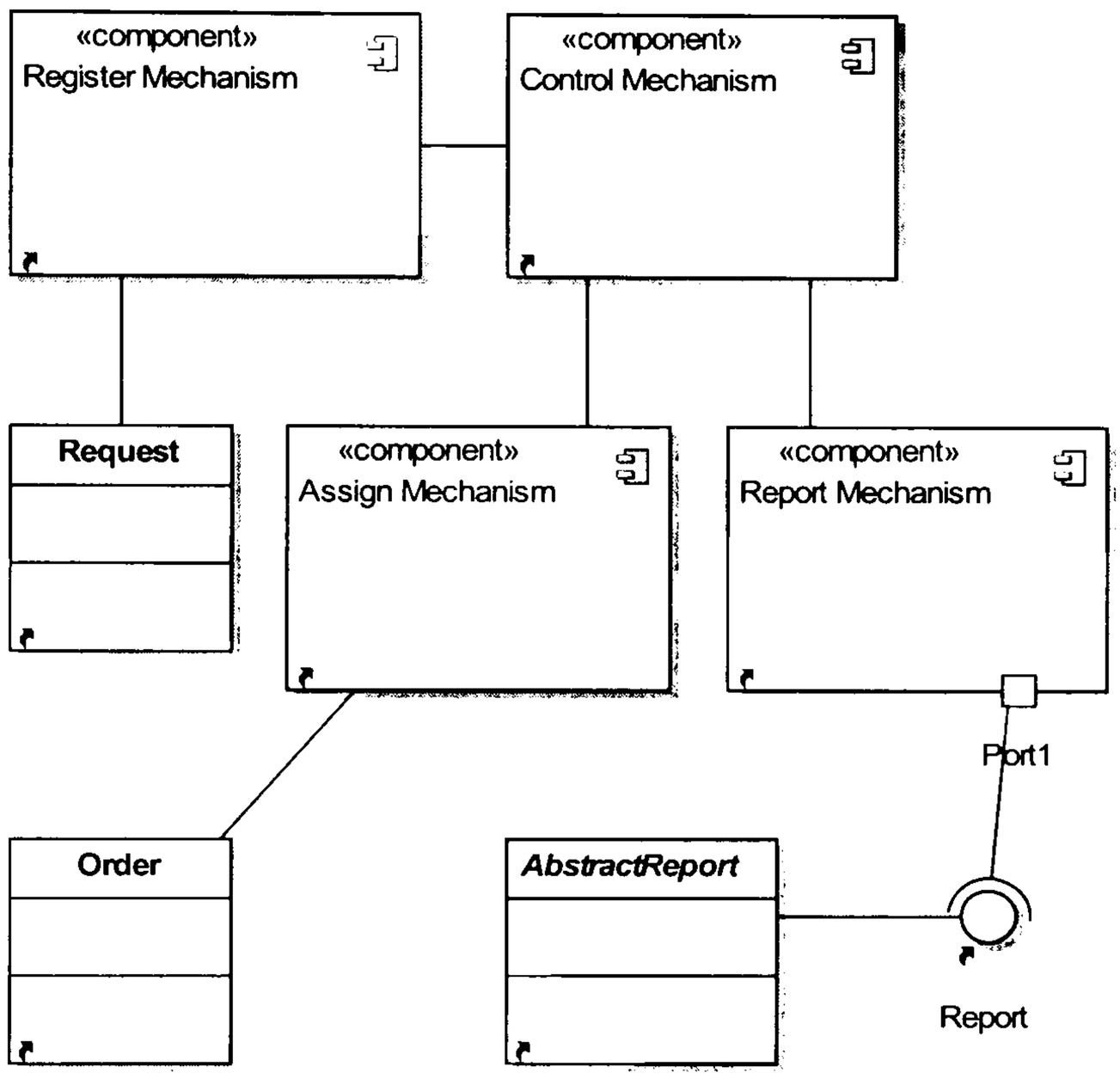

Figure 8-1-2 RAT Domain Pattern

- The subsystem Registration communicates with the external stakeholder systems and produces an internal Request entity that must be tracked in the system.
- The subsystem Assignment communicates with the external stakeholder system and assigns actions to be taken on the request.
- The subsystem Presentation communicates with the external stakeholder system and presents to authorised stakeholders of relevant information on request status.

The RAT domain pattern is shown in Figure 8-1-2.

8.1.3 Manufacturing Systems

A MAN instance creates a product from raw materials. This is the core process and its activities are:

- Process and check raw materials
- Convert raw materials to 'half-products'
- Package and dispatch half-products.

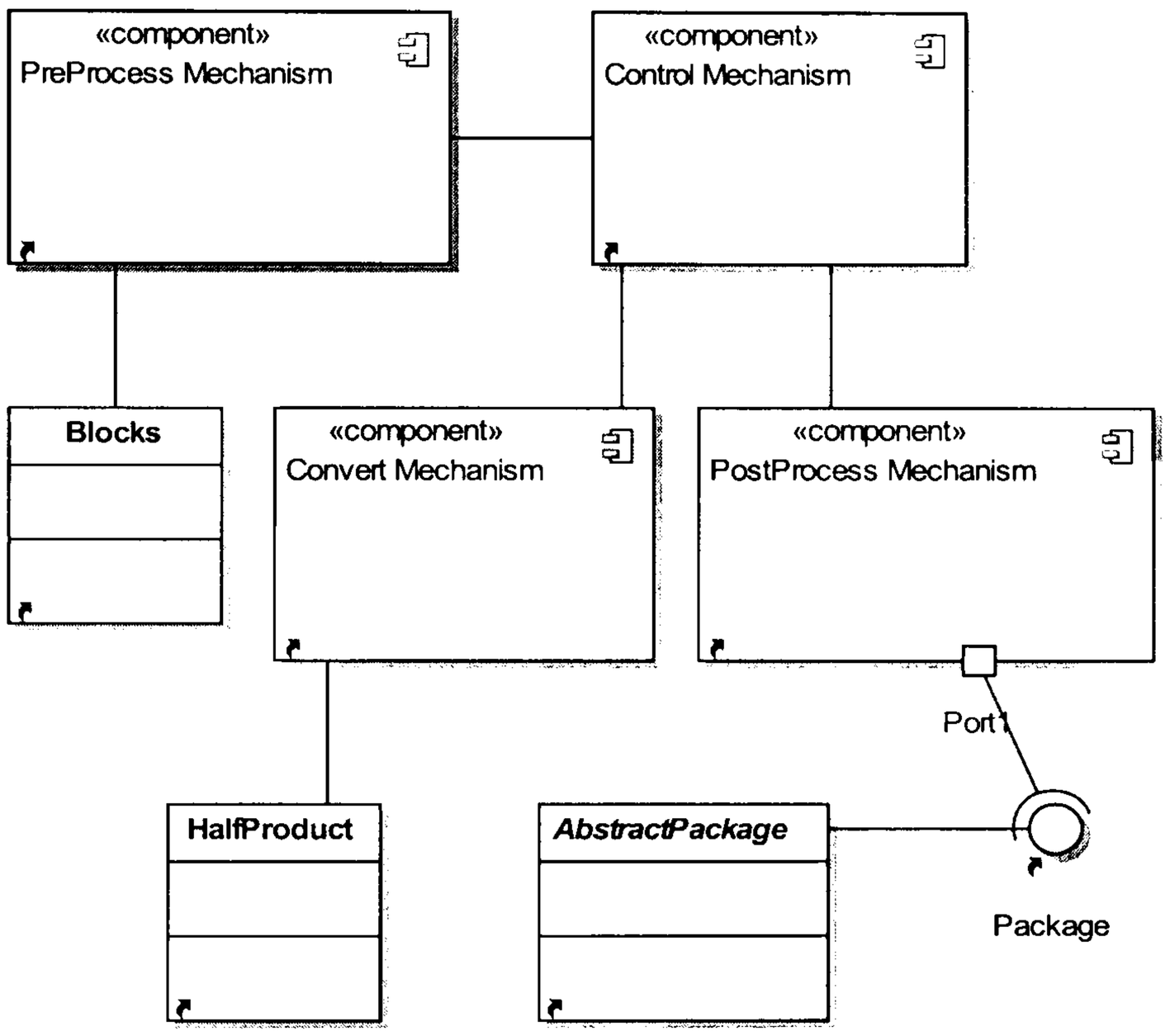

Figure 8-1-3 MAN Domain Pattern

The architecture of MAN is shown in Figure 8-1-3. The pre process subsystem takes basic blocks for conversion subsystem to build half product that is packaged by post process subsystem to deliver to specific clients.

8.1.4 Access Control Systems

There are two main processes in ACS systems:

- Authentication: securely identifying principals

- Authorisation: controlling which principals can execute which operations on which resources.

The main activities in the Authorisation process are:

- Accept a request from a subject to gain access to an object

- Check whether access is allowed

- If successful, execute the request on behalf of the subject.

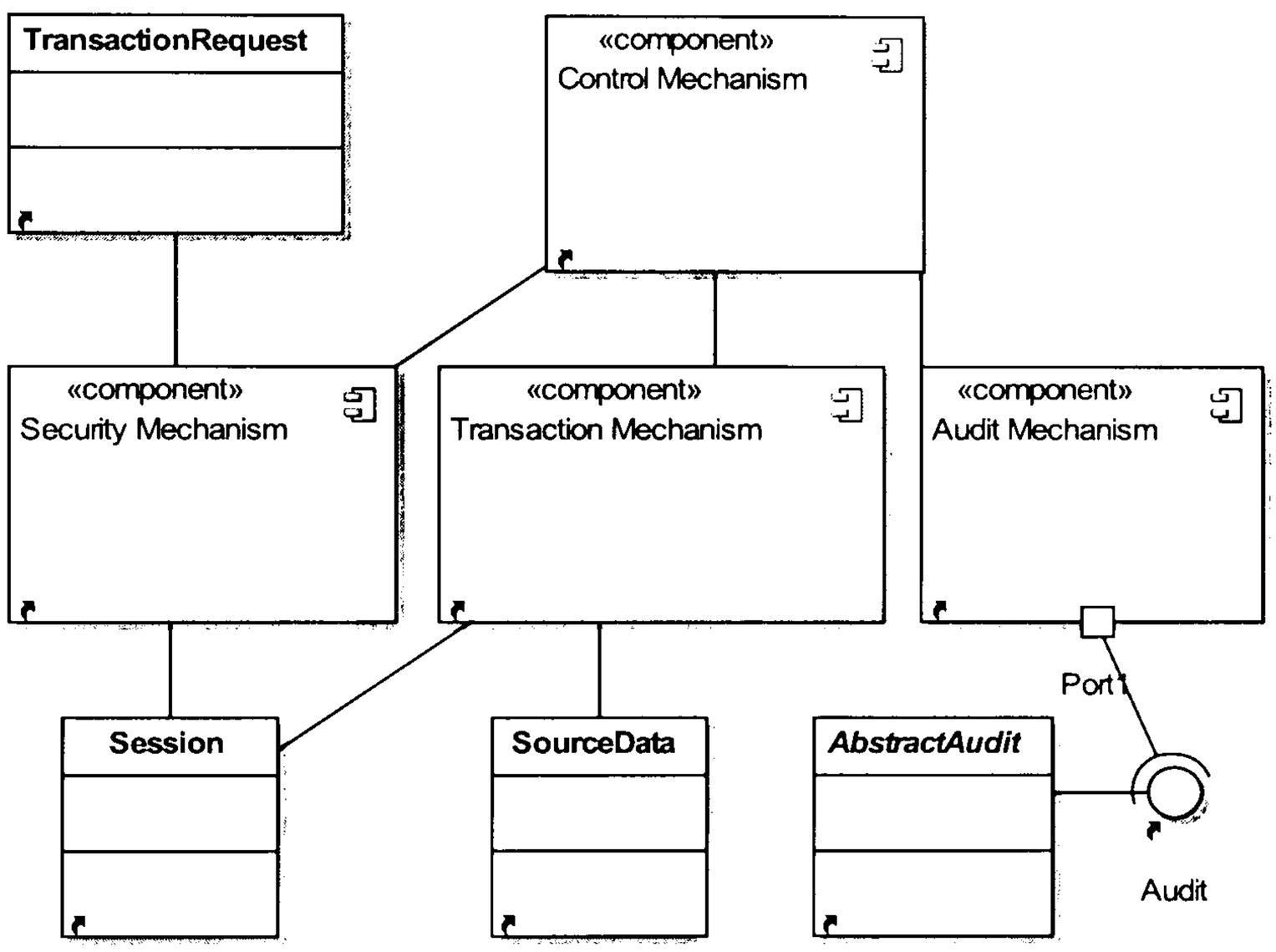

Figure 8-1-4 ACS Domain Pattern

The architecture of ACS is shown in Figure 8-1-4, where security subsystem accepts transaction request and then check the section associated with the request for the authorisation and authentication. If the request passes the checking, transaction subsystem will execute the request. During the whole process, the audit subsystem records any information defined in specific audit objects.

8.1.5 Lifecycle Model

The systems in this category consist of three components, namely a MAN instance, a RAT instance and a MIS instance. Lifecycle Models (LCM) are very important because most real world applications are composed of multiple lifecycle models.

8.2 System Design Patterns

While domain architectural patterns help top-down reverse engineering by mapping domain concepts to software system, the analysis of design patterns that establish strong relationships between software objects can expose much more details of an existing system. Some design patterns that are widely used in Web-based systems are classified into five categories.

8.2.1 Web Presentation Patterns

8.2.1.1 Model-View-Controller

Model-View-Controller (MVC) architecture is based on an older graphical user interface (GUI) design pattern that has been around for some time. However, many of the same forces behind MVC for GUI development apply to Web-based systems development. Figure 8-2-1 shows the MVC system design pattern.

Data retrieval and display is the key functionality of most Web-based systems. Any changes made from the user interface to the data are kept in a data store by the system and the key flow of information is between the data store and the user interface. Thus, by tying the two pieces together, the amount of coding can be reduced and application performance can be improved.

However, one problem of this solution is that the user interface is changed much more frequently than the data storage system. Coupling the data and user interface pieces makes it difficult to modify the individual parts.

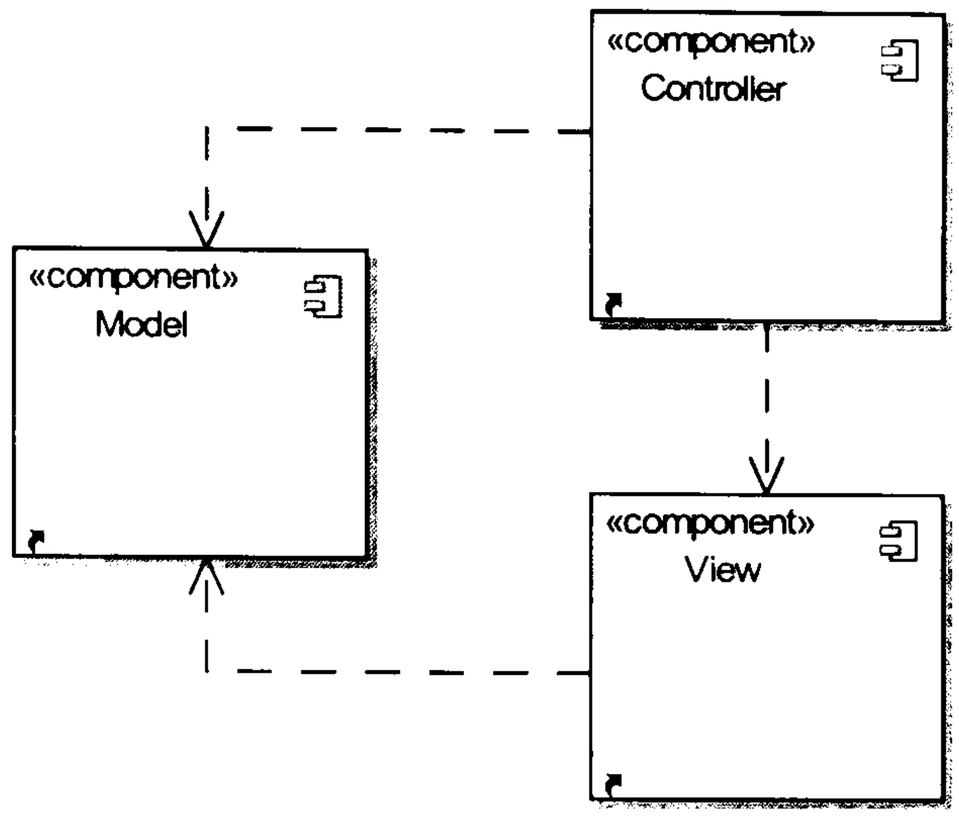

Figure 8-2-1 MVC System Design Pattern

The Model-View-Controller (MVC) pattern separates the modeling of the domain, the presentation, and the actions based on user input into three separate classes [Burbeck92]:

- Model Components. Model components provide an interface to the data and/or services, via which Controller components rely on the model components to perform data access and manipulation. Thus, the model component encapsulates the business logic. Model components can be implemented in various forms such as Java beans, Enterprise JavaBeans (EJBs) or Web services.

- View Components. View components are responsible for generating responses to client browser. However, other view technologies such as WML can be used as well and changing the implementation of view component will not have any impact on the Model of a Web-based system.

- Controller Components. At the core of the MVC architecture are the controller components. A Controller receives requests for the application and controls the way that the Model and View layers interact by directing the data flow between them.

The MVC system design pattern is shown in Figure 8-2-1. Chapter 5 Web Application Framework discusses the details of MVC concepts with a proposed new MVC

framework for evolvable Web applications. Popular MVC frameworks are compared in 3.3.

8.2.1.2 Page Controller

Using MVC pattern can separate the user interface components of dynamic Web application from business logic. While the Web pages are constructed dynamically, the navigations between them are mostly static. To achieve reuse and flexibility, Controller needs to be structured to avoid code duplication.

Page Controller pattern accepts input from requests, invokes the requested actions on the model and determines the correct view for displaying results. The dispatching logic is common for most page controllers. Separating this logic from any view-related code can be achieved by creating a common base class for all page controllers, which avoids code duplication and increases consistency and testability. Figure 8-2-2 shows how the page controller relates to the model and view.

Defining separate page controllers isolates model from the specifics of the Web request such as session management, the use of query strings or hidden form fields to pass parameters to the page. By creating a controller for every link in the Web application, controllers are kept simple and needs to tackle only one action at a time. Figure 8-2-2 shows Page Controller system design pattern.

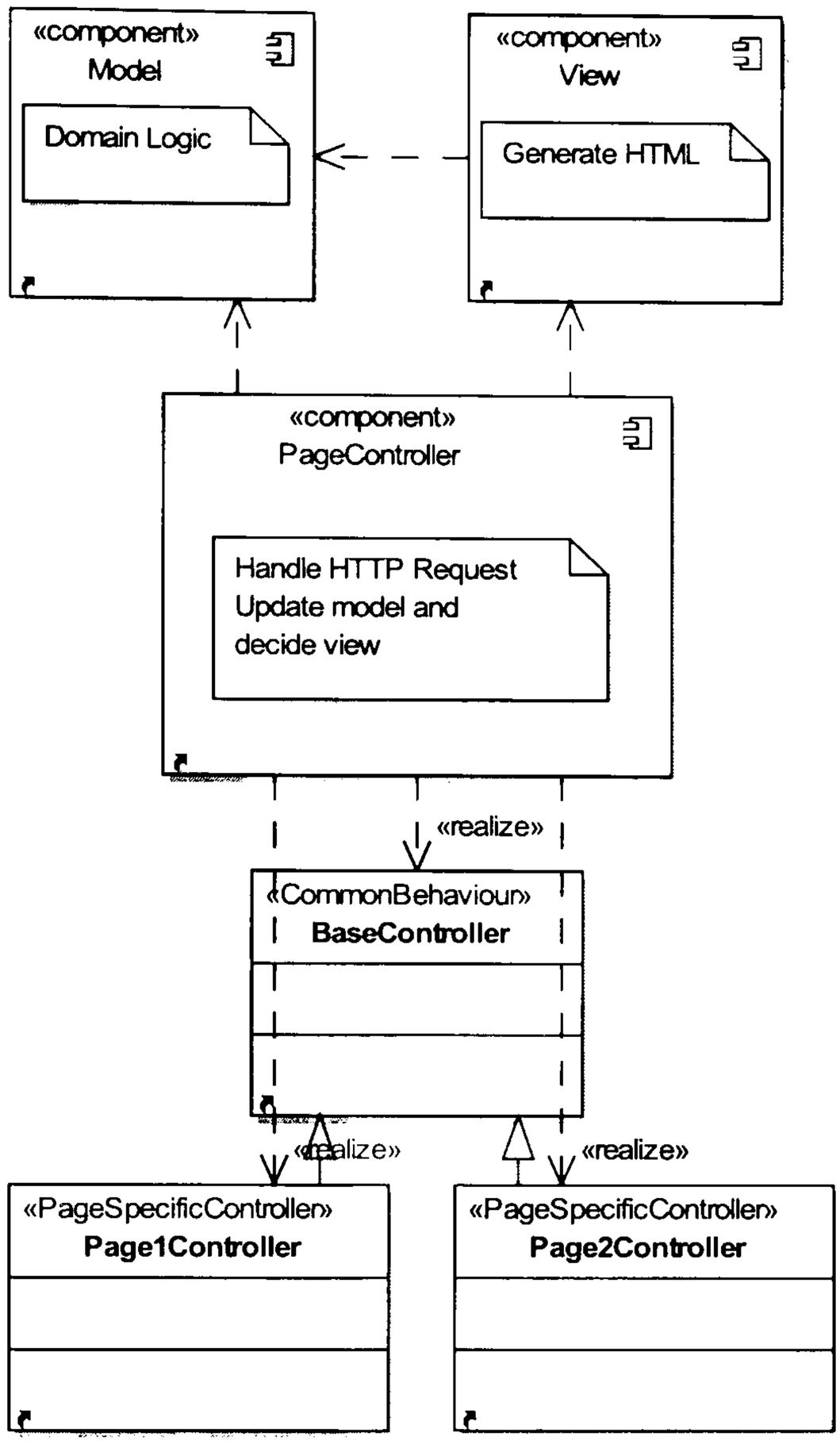

Figure 8-2-2 Page Controller System Design Pattern

8.2.1.3 Front Controller

Using MVC pattern can separate user interface logic of dynamic Web application from the business logic. Using Page Controller pattern can reduce duplicate code in Controllers. However, page controller classes can have complicated logic as part of a deep inheritance hierarchy and the navigation between pages is often specified dynamically based on configurable rules.

Front Controller is introduced to structure a controller for very complex Web applications to achieve reuse and flexibility while reducing code duplication. Front Controller addresses the decentralisation problem present in Page Controller by

channelling all requests through a single controller, which is usually implemented in two parts: a handler and a hierarchy of commands. Figure 8-2-3 shows Front Controller system design pattern.

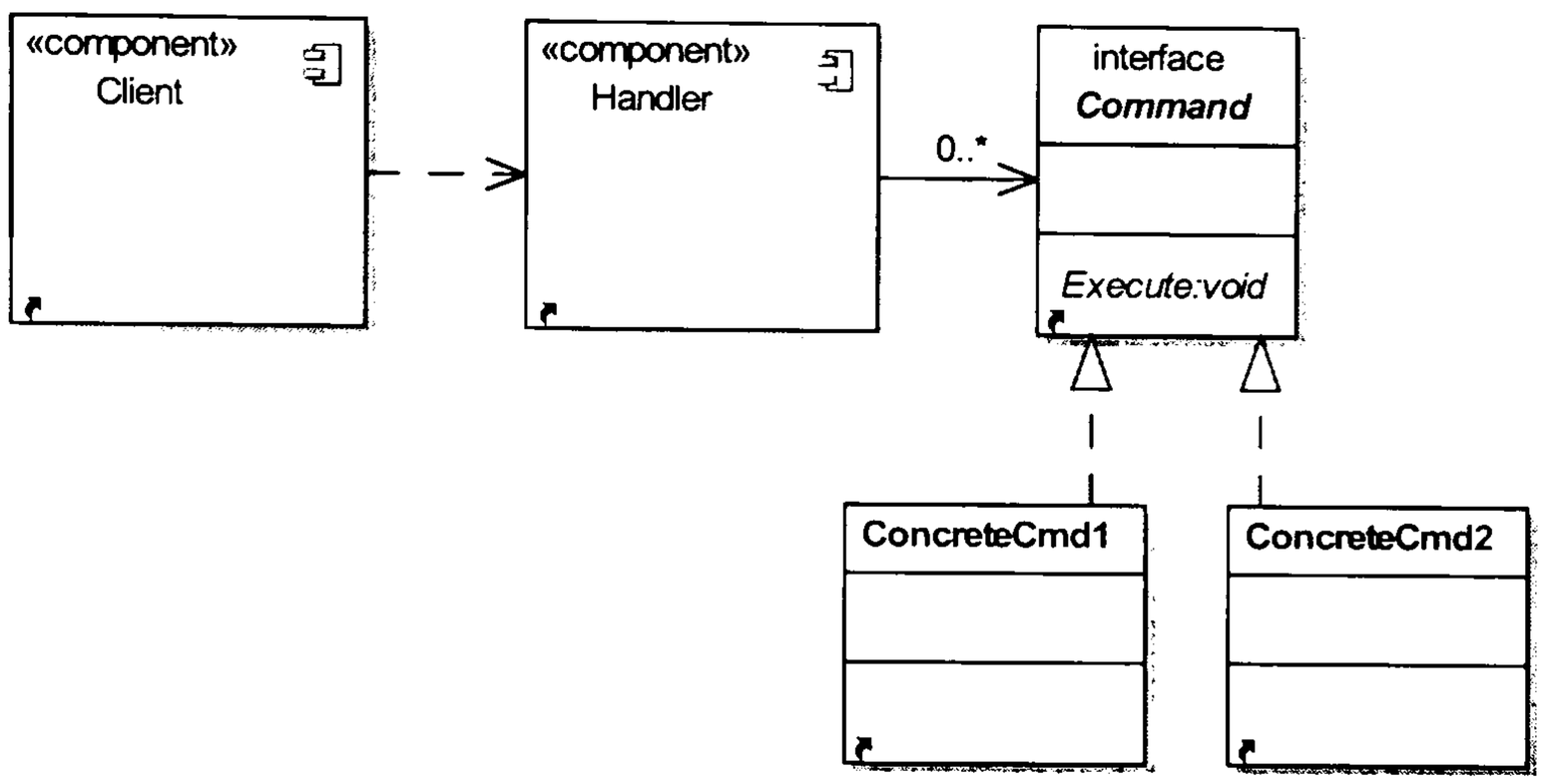

Figure 8-2-3 Front Controller System Design Pattern

The handler has two responsibilities:

- Retrieve parameters. The handler receives the HTTP requests with relevant parameters from Web server.
- Select commands. The handler uses the parameters from the request to execute corresponding command.

Front Controller pattern is very important for building an evolvable Web Application Framework that is discussed in detail in Chapter 5.

8.2.1.4 Intercepting Filter

Intercepting filter pattern allows developers to build pre- and post-processing steps around Web page requests by creating a set of filters that can be chained together to execute a set of common processing steps before the page request is passed to the controller object.

Filters implement identical interfaces and they do not have explicit dependencies on each other. Thus, new filters can be added without affecting existing filters [Gamma95].

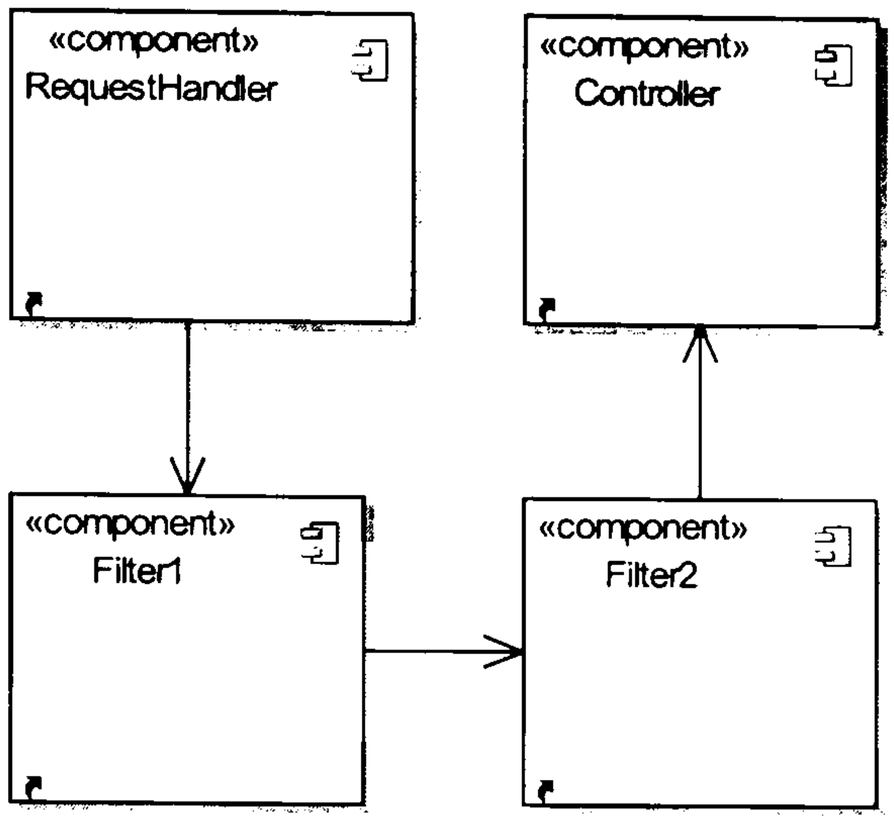

Figure 8-2-4 Intercepting Filter System Design Pattern

Filters should be designed in such as way that they have no dependencies on other filters and make no assumptions of the order in which the filters are executed. Intercepting Filter system design pattern is shown in Figure 8-2-4.

8.2.2 Distributed Computing: Instance-based Patterns

Patterns in this category are dedicated to the concurrency and synchronisation issues raised in distributed computing, which is powerful but is based on an inherently unreliable network.

Instance-based patterns extend the model of object-oriented computing across network boundaries, where the remote objects are regarded the same as the local ones. They simplify the development of distributed computing at the expense of a complicated interaction model and tight coupling between consumer and provider. Another category of distributed computing patterns which address this issue will be given later in this chapter.

8.2.2.1 Broker

Software systems are distributed via networked computers for various reasons.

- Distributed systems can take advantage of the computing power of multiple CPUs or a cluster of low-cost computers.

- Required software, services or resources may only be available on specific computers or via Internet.

- Parts of a software system may have to run on different network segments due to security management.

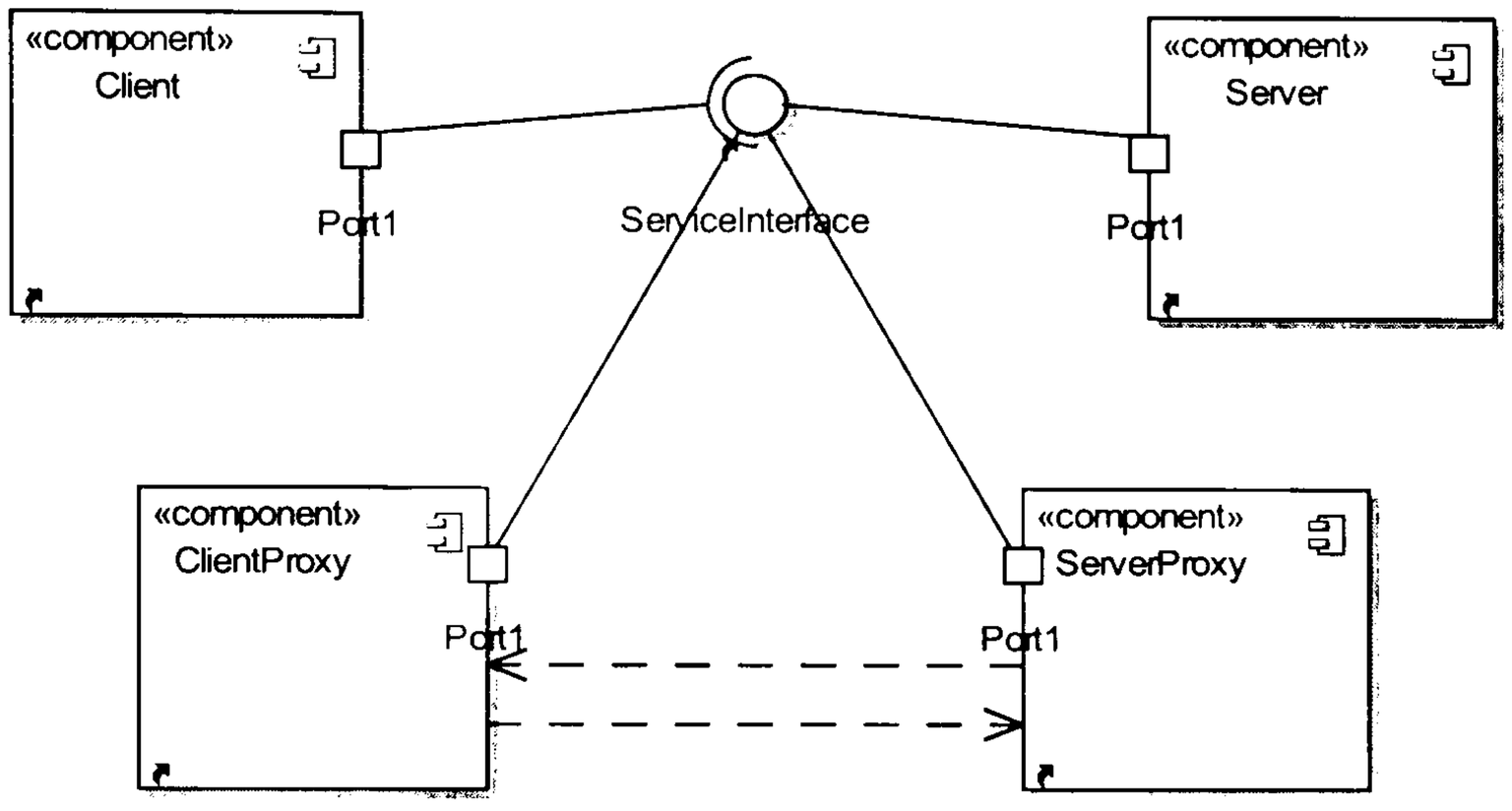

Figure 8-2-5 Broker System Design Pattern

Implementing a distributed system needs to address issues such as concurrency, cross-platform connectivity, and unreliable network connections. Broker pattern is used to structure a distributed system in order to hide the implementation details of remote communication from developers, by encapsulating them into a layer other than the business component itself [Buschmann96].

Broker allows a client to invoke remote methods defined in an interface just as it would invoke any local ones. The methods inside the client interface trigger services to be executed on remote objects, which is transparent to the client because the same interface is implemented by the remote service object. Figure 8-2-5 shows the Broker pattern.

8.2.2.2 Data Transfer Object

In distributed environment, making multiple remote calls for a single client request will increase response time. Data Transfer Object (DTO) preserves the simple semantics of a procedure call interface without having the latency issues inherent in remote communication.

DTO holds all data required for the remote call. A remote method is defined such that the only parameter is a DTO and the return value another DTO which is returned to the calling application and stored as a local object. In this way, a client application can make a series of individual procedure calls to the DTO without incurring the overhead of remote calls. DTO is described in [Fowler03].

8.2.2.3 Singleton

Sometimes a certain type of data must be unique and available to all other objects in the system. This can be useful for applications that require a single point of entry for a block of functionality.

Singleton pattern instantiates one instance of the class the first time it is called, and maintains that single instance throughout the lifetime of the application. Once it is instantiated, all future requests will be through a single global point of access. A Singleton pattern is created as follows.

- A public static method needs to be created as an access point for other objects.

- This public static method returns a reference to the instance if the instance has been created. Otherwise it creates the instance of its own and returns the reference to it.

- The class constructor needs to be declared as private so that no other object can create a new instance without invoking the public static method.

Figure 8-2-6 shows the static structure of Singleton pattern, which consists of a simple class holding a reference to a single instance of its own.

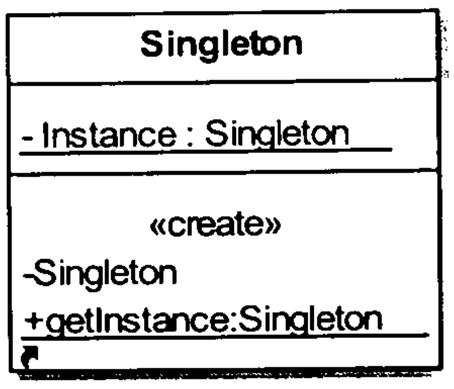

Figure 8-2-6 Singleton System Design Pattern

8.2.3 Distributed Systems: Service-based Patterns

Patterns in this category are dedicated to structuring a custom-developed solution in a service-oriented environment. These patterns help developers to expose application functionality as services while hiding the details of consuming services exposed by other applications

8.2.3.1 Service Interface

Enterprise applications often have functionality that must be available across a network and accessible to various types of systems. Interoperability is the key to such design requirements. In addition to various systems, different types of communications protocols and operational requirements have to be accommodated too.

Service Interface pattern makes application functionality available to other applications, while ensuring that the interface mechanisms are decoupled from the application logic. This is achieved by designing an application as a collection of software services, each of which provides a service interface used by consumers of the application to interact with the provided service.

Figure 8-2-7 shows a service gateway consuming a service provided by a service interface. The collaboration between these two components is governed by a contract.

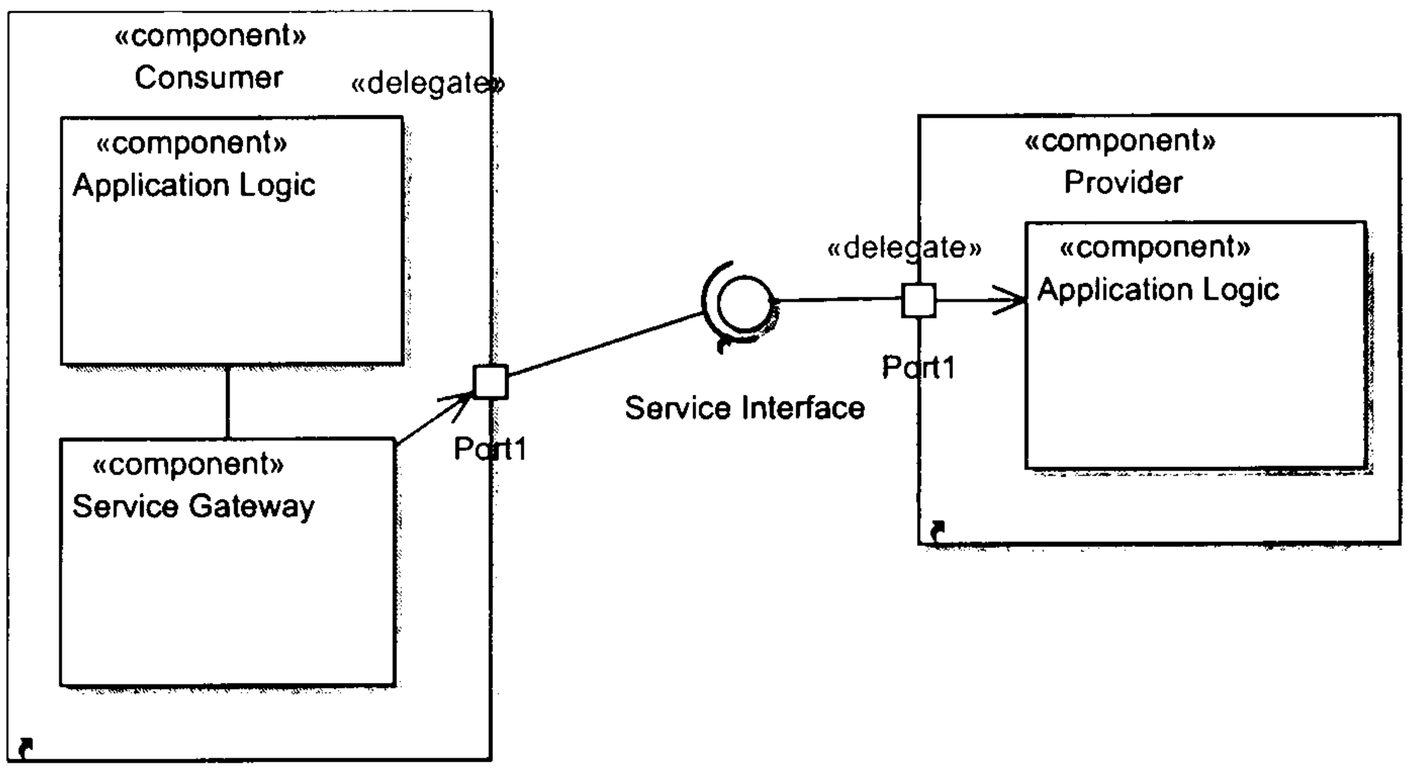

Figure 8-2-7 Service Interface System Design Pattern

Service Interface provides an entry point to the functionality exposed by an application. It decouples the application's business logic from the details of its communication mechanism with other systems such as network protocol, data formats and security.

8.2.3.2 Service Gateway

A service provided by an application defines a contract that must be obeyed by consumers accessing the service. The contract defines details of the communication mechanism such as the technology, communications protocols and message definitions. All consumers of a service must fulfil its responsibilities specified in the related contract. Service Gateway pattern decouples the details of fulfilling contract responsibilities specified in a service from the rest of the application.

Service Gateway pattern encapsulates the code implementing the consumer part of a contract into a separate Service Gateway component. Service gateways act as proxies to other services, hiding the details of connecting to the source.

Service Gateway derives from Martin Fowler's Gateway pattern [Fowler03] and is adapted for service-oriented architectures to encapsulate a consuming application's access to services. Service Gateway pattern is often combined with Remote Facade pattern [Fowler03] to decouple a consumer application from its service providers. Remote Facade encapsulates complex functionality in service providers and exposes that functionality with a single access point to consumer applications. Service Interface pattern is a specific type of Remote Facade adapted for service-oriented architectures.

Service Gateway component hides the low-level details of communication mechanism such as communications channel, data formats, service discovery, process adapter and calling semantics (synchronous or asynchronous).

8.3 Integration Patterns

Enterprise integration aims to coordinate heterogeneous systems to work together to achieve a unified set of functionality. These heterogeneous systems include both legacy and modern systems, which are running on various platforms on different geographical areas. Some systems do not have built-in integration mechanism to work with other

systems and sometimes it could be very difficult, especially for legacy systems, to add such facility by pure reengineering techniques. Integration Patterns are solutions to those issues.

The following are some criteria for designing an Integration Pattern [Micro04].

- Integrated applications should have their dependencies on each other minimised so that each can be modified without interfering others.

- Changes to the application and the amount of integration code should be minimised during integration.

- The format of the data exchanged between integrated applications must be agreed on between them. If changing existing applications to use a unified data format is difficult or impossible, an intermediate translator can unify applications with different data format requirements.

- The length of time for data sharing between integrated applications should be minimised, which can be achieved by exchanging data frequently and in small chunks with latency examined to evaluate the efficiency. The receiver applications should be informed as soon as shared data is ready in order to reduce out of sync.

- In addition to data, functionality can be shared as well, which can provide better abstraction between applications. However invoking remote functionalities may have significant consequences to an integration plan.

- Remote connections are slow and less reliable than a local function call. Asynchronous communication allows the source application to continue other work, not having to wait for the remote application to return the result. However such a solution is also more complex to design, develop and debug.

It is not easy to design an integration pattern that meets all criteria equally well. Various pattern for integrating applications have to be used in different situations. They can be classified into four main categories.

8.3.1 File Transfer

File Transfer integration pattern has each application produce files of shared data consumed by others and consume files produced by others. Figure 8-3-1 shows File Transfer Integration pattern.

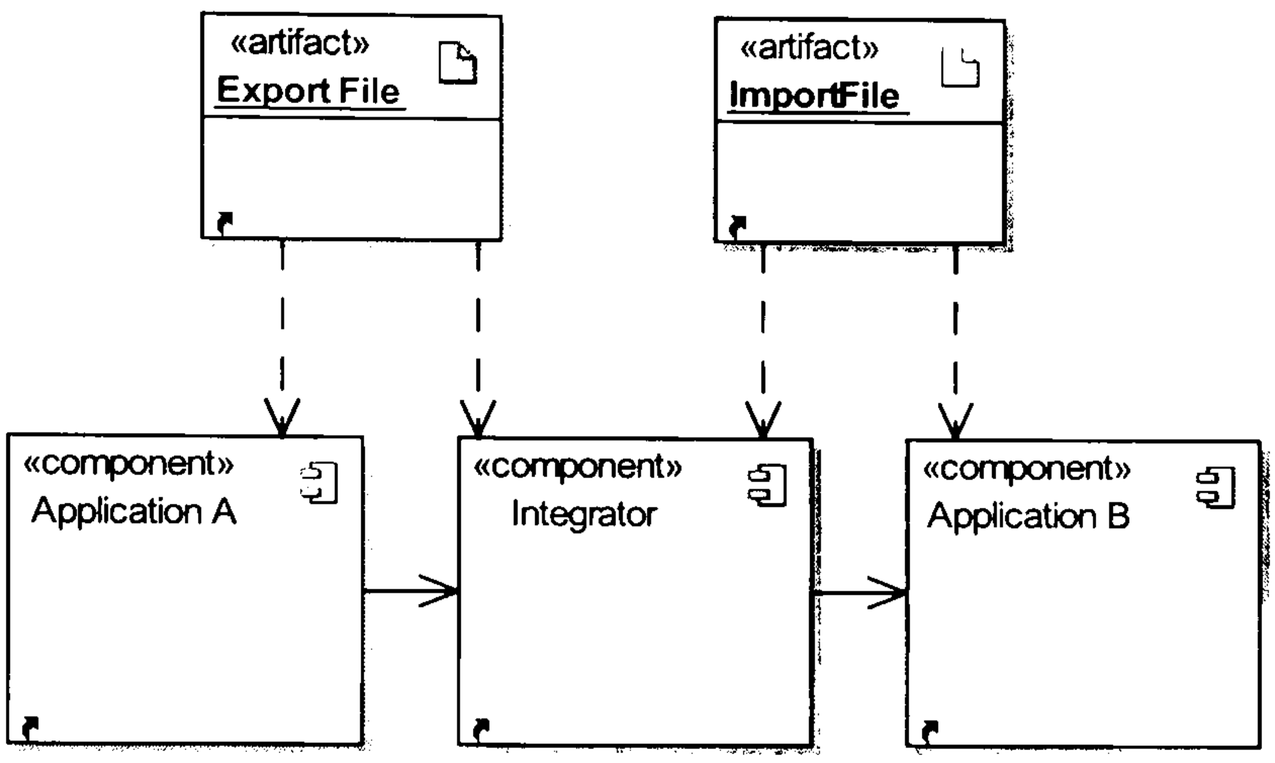

Figure 8-3-1 File Transfer Integration Pattern

Using File Transfer integration pattern, integration can be achieved without exposing the details of the internals of an application, which means the integrated applications are quite independent from each other. Making internal changes to one application will not affect others as long as it produces the same data in the same format. The files can be regarded as the public interface for each application. The requirements of Fire Transfer integration pattern are given as follows.

- Integrated applications must agree on file-naming and directory conventions.

- Integrated applications must agree on the details of operations on the shared files, such as which application will delete the files and when to delete them.

- Integrated applications need to prevent file sharing violation, where one application is trying to read a file being written by another application.

- The access to shared files should be guaranteed by transferring the files to accessible places for applications having no limited access compared with others.

8.3.2 Shared Database

Shared Database integration pattern has applications store the shared data in a common database. For applications that are being built independently with various languages and platforms, Shared Database integration pattern allows information to be shared in a fast and consistent way.

File Transfer pattern allows applications to share data. However it can lack timeliness which is often critical for system integration. In addition, File Transfer may not enforce data format effectively because of incompatible ways of observing the data. Figure 8-3-2 shows Shared Database integration pattern.

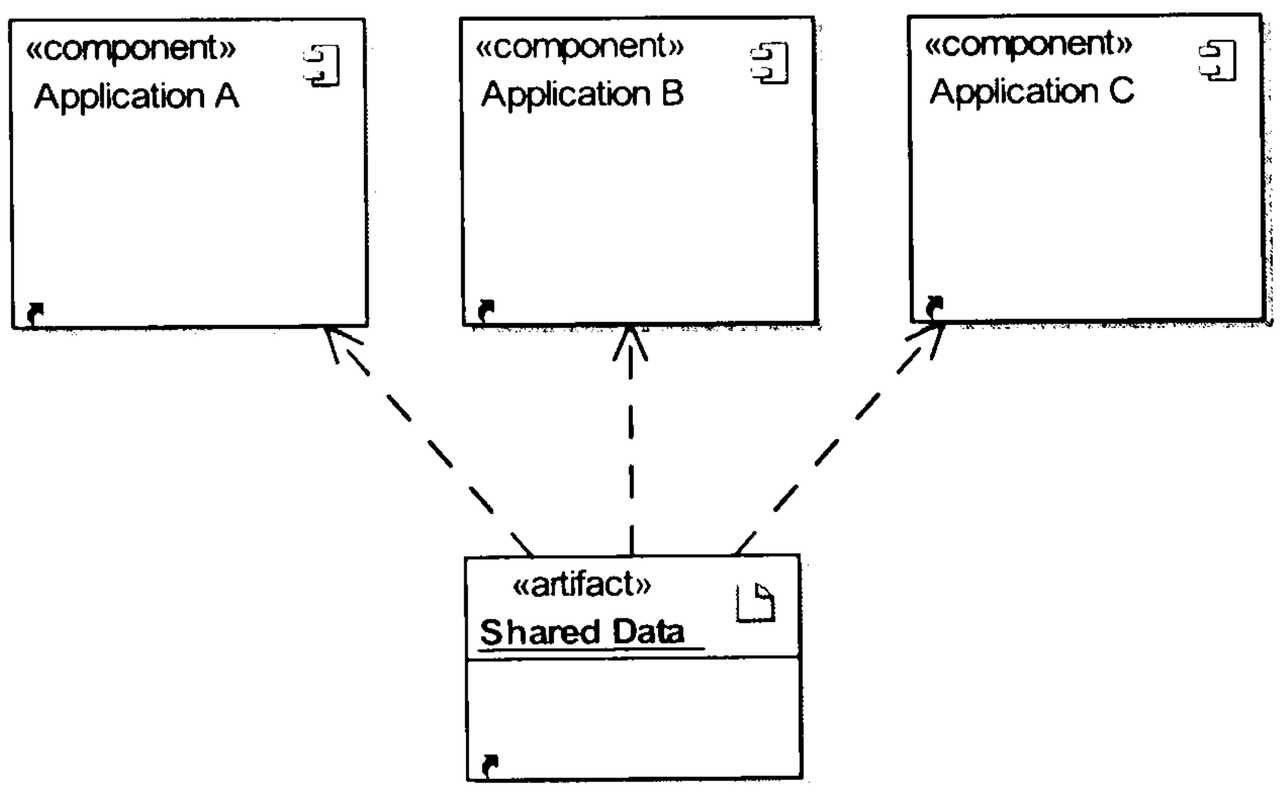

Figure 8-3-2 Shared Database Integration Pattern

Compared to File Transfer, Shared Database integration pattern provides a central, agreed-upon datastore accessible to all applications. Shared Database prevents semantic dissonance and is well supported by most development platforms for the application of SQL-based relational databases.

8.3.3 Remote Procedure Invocation

Remote Procedure Invocation integration pattern has each application expose some of its procedures to be invoked remotely by other applications to initiate behaviour and exchange data. Figure 8-3-3 shows Remote Procedure Invocation integration.

File Transfer and Shared Database allow data to be shared by applications. However sharing data is often not enough and changes in data often require actions to be taken across different applications.

Remote Procedure Invocation applies the principle of encapsulation, where an application makes a call to other application to get information owned it or modify its data. This allows each application to keep the integrity of its data. In addition, each application can change the format of its internal data without interfering others.

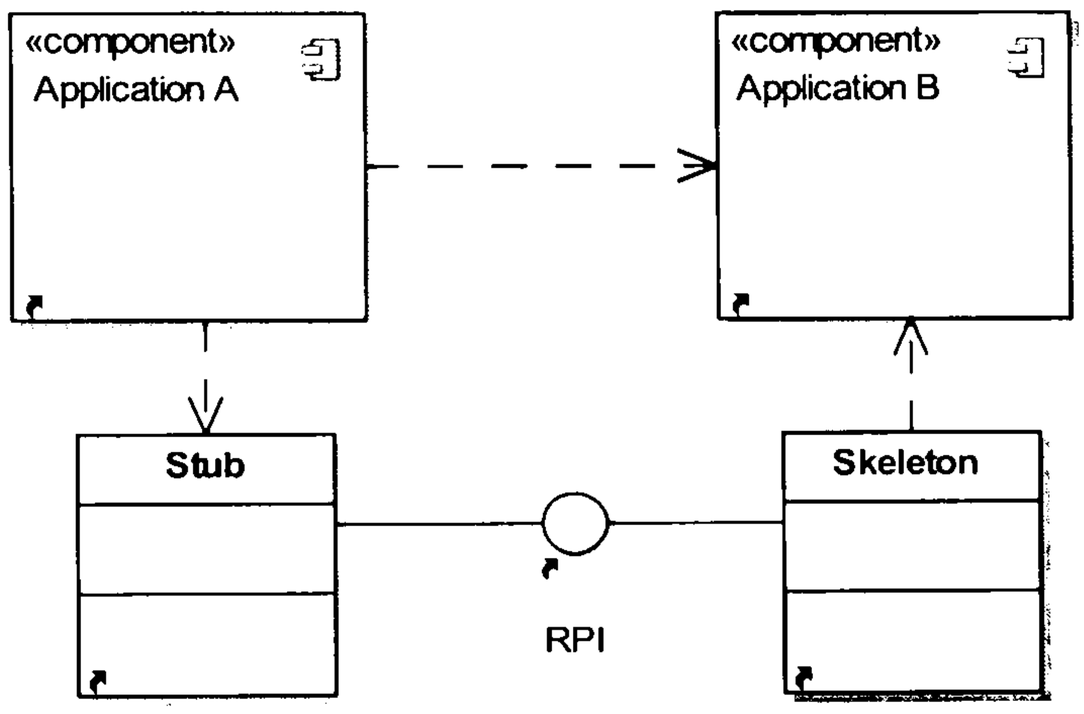

Figure 8-3-3 RPI Integration Pattern

Using RPI could also be problematic for performance and reliability and can lead to slow and unreliable systems [Waldo94 and EAA03]. In addition, the remote calls that each system supports tend to couple different systems due to its synchronous behaviour.

8.3.4 Messaging

Messaging integration pattern has each application connect to a common messaging system to exchange data and invoke behaviour via messages.

File Transfer and Shared Database allow data to be share by applications but not functionality. Remote Procedure Invocation allows applications to share functionality with the applications tightly coupled by synchronous communication.

Messaging integration pattern hides the details of any disk structure or database for storing the data from the applications. In addition, Messaging integration pattern can

produce and transfer a number of small data packets quickly, while the receiver application is notified when a new packet is available for consumption. Figure 8-3-4 shows Messaging integration pattern.

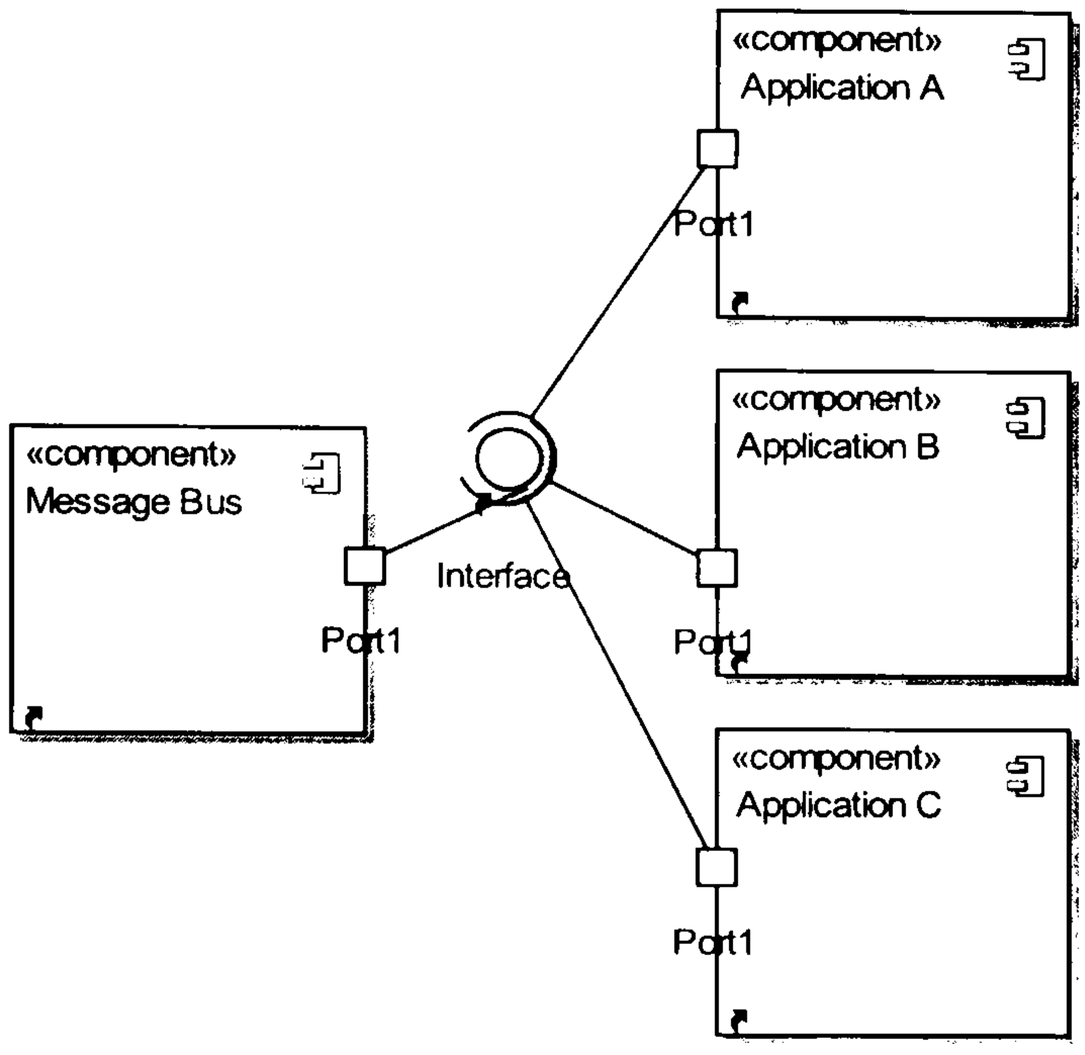

Figure 8-3-4 Messaging Integration Pattern

Messaging integration pattern allows transferring packets of data frequently, immediately, reliably and asynchronously in customisable formats.

8.4 Summary

This chapter discusses classification of patterns that exist at different levels of abstraction during the evolution of a software system.

- Domain Architectural Patterns. A domain architectural pattern stands for the abstraction of a set of applications with similar functionality, behaviour and structure. It describes the essential features in some business domain. Domain architectural patterns are used in constructing concept model (4.4.1) and architecture migration (4.4.7). They can be divided into five categories.

- System Design Patterns. While domain architectural patterns help top-down reverse engineering by mapping domain concepts to software system, the

analysis of design patterns that establish strong relationships between software objects can expose much more details of an existing system. Some design patterns that are widely used in Web-based systems are classified into five categories.

- Integration Patterns. Integration patterns are used to coordinate heterogeneous systems to work together to achieve a unified set of functionality. These heterogeneous systems include both legacy and modern systems, which are running on various platforms on different geographical areas. Some systems do not have built-in integration mechanism to work with other systems and sometimes it could be very difficult, especially for legacy systems, to add such facility by pure reengineering techniques. Integration Patterns are solutions to those issues.

CHAPTER 9

XML Transformers for Evolvable Data Management

Data management is of significant importance to Web-based systems evolution, and XML is the key to enterprise data management. For modern Web-based systems, especially enterprise applications, XML is the infrastructure for specifying document storage, transaction protocol and message communication. For legacy Web-based systems, one of the most popular ways to "update" them, after reverse engineering, is to apply up-to-date XML data management technologies to facilitate future maintenance and integration. This chapter analyses the roles of XML in data management for Web-based systems evolution.

9.1 XML-based Data Management

Data Management (DM) aims to provide a unified solution to organising and utilising organisational data resources effectively and efficiently to the maximum extent.

DM is applied in different contexts, from high-level architectural disciplines to low-level implementation details. It emphasises on standardisation of data naming and usage conventions, pattern or trend discovery data abstraction and analysis, data storage and formatting in various contexts and data quality evaluation. Most of the techniques applied in DM are equally powerful when applied in software evolution projects, especially for Web-based enterprise systems which are inherently data-centric.

The data-centric nature of Web-based enterprise systems requires software development to be carried out with focus on DM, while XML is widely used in modern application development. It is reasonable to combine the two in a unified way to software development and evolution.

In addition to data representation, XML enables the management of unstructured data, which could not be handled by traditional DM capabilities. Unstructured data is one of

challenges faced by reengineers when modernising a legacy system. There are various ways to represent and integrate such data resources. The features of XML can help to build such a solution that is capable of accommodating future changes as well as integrating existing resources, either structured or not.

This rest of this section summarises the characteristics of XML-based data management.

9.1.1 Standardisation

The first level of standardisation comes from XML itself. XML is designed as a platform independent and open standard, which is pushed by the World Wide Web Consortium (W3C) towards vendor-neutral standards and technologies for interoperability over the Internet.

The XML standard is language and environment independent, where XML and related standards can be applied regardless of the chosen language or environment for an organisation. An enterprise system built with Java and J2EE architecture can make use of XML-related technologies as well as a Microsoft .Net based system. The separation of XML standards and underlying infrastructure provides huge flexibility for organisations to choose implementation technologies both in development and maintenance.

The second level of standardisation comes from the industry application of XML. XML specifies text Markup rules, but allow enterprises to create their specific vocabularies. XML Schema helps create new vocabularies in a convenient way and the creation of both industry-specific and cross-industry standards have been growing prosperously.

9.1.2 Self-Descriptiveness

In the world of Web and Internet, the explosion of resources requires information to be self-descriptive for management. XML makes use of tags as descriptive elements and attributes to identify information characteristics such as name, type, location or position. Such information is intuitive and self-explanatory to humans, while formal and flexible to machines. An XML document representing information such as a delivery address including address lines, city, county, country and post code can be easily identified by

humans. On the other hand, automated applications can parse the XML document, test elements and attributes and process the contained information via parsers. Listing 9-1-1 shows the XML document for customer information, where address lines, city, county and other parts of an address are described by their element names.

```
<?xml version="1.0" encoding="UTF-8"?>
<Customers>
 <Customer CustomerID="C012345" CustomerType="Home and Home Office" Source="List 01-
02-03">
  <Addresses>
   <Address AddressType="Permanant" AddressNo="1">
    <Lines>
     <Line LineNo="1">Room 12, Court 23</Line>
     <Line LineNo="2">Riverside Road</Line>
    </Lines>
    <City>Nottingham</City>
    <Region>
     <EastMidlands CountyName="Nottinghamshire">Nottinghamshire</EastMidlands>
    </Region>
    <PostalCode>NG13</PostalCode>
    <Country CountryCode="UK" CountryNo="044">UK</Country>
   </Address>
  </Addresses>
 </Customer>
</Customers>
```

Listing 9-1-1 Customer Information Document

It is a significant advance to describe the contents of a transaction in XML instead of more traditional transaction and interface file formats, for which the raw data are stored with fixed location within the file or pre-decided separators to differentiate between data items. The traditional way of handling has many limitations.

- The lack of standard navigation mechanism for applications to identify, extract and process the contained data. This could have a potential impact on system evolution in terms of problem-solving, improvement and enhancement.

- Analysis of traditional transaction files requires cross-referencing to the file structure.

- The lack of self-descriptive notation for traditional interface file makes it difficult to understand the context of the contained information.

In addition to the structure of XML document, the use of XML schema is another advantage of XML for further describing the contained information. XML Schemas enables the description of data type as well as other rules and constraints.

- Base and derived data types , such as numeric, string, date and boolean
- Extended data types, such as custom data types
- Facets, such as length and fractional digits
- Value limits, such as minimum and maximum values
- Enumeration, such as a set of allowable values for an element or attribute
- Element repeating occurrences, such as cardinality and modality
- Patterns, such as format and edit patterns

Unlike Hypertext Markup Language (HTML), XML allows the application of enterprise naming standards or taxonomy in designing XML elements and attributes, which are predefined in HTML and can not be easily extended.

XML-based architecture description is discussed in Chapter 7, where a set of XML schemas are defined for representing the architecture of Web-based systems.

9.1.3 Separation of Concerns

The elements and attributes of XML represent metadata used in definitions for actual content, which applies the principle of separation of information from its presentation. Unlike HTML, XML allows to define content that can be used in various ways including formatting for Web browser of different devices, passing XML messages for processing and transforming the information in various presentation formats, such as PDF, HTML, or Microsoft Word. In addition, the metadata can be utilised by intelligent search engines to identify specific information.

In Web-based systems evolution, the application of XML, however, is not to replace completely HTML documents. HTML was created for content presentation in client browser. Its elements and attributes are dedicated to document formatting and user interactive actions such as collecting user input information from the browser and submitting it to a server. HTML should be used as a protocol for information representation and collection, but not for information storage. Keeping information in HTML documents tightly couples the content and its representation, which makes it difficult for information reuse and maintenance.

The weakness of HTML in information storage is overcome by the use of XML, which is ideal for keeping data. Many Web-based systems can dynamically produce client side HTML pages from XML documents via eXtensible Stylesheet Language Transform (XSLT), which is part of XML technology and is used to format or transform XML content from one form to another. For example, java servlet can transform an XML document using XML stylesheet to HTML page(s) to be displayed in a browser.

9.1.4 Interoperability

The interoperability of XML comes from its Unicode-encoded files and the high availability of XML parser tools for most mainstream platforms. The issues in heterogeneous computing environment, such as character set conversion, are addressed by XML. Special support for simple ASCII or complex character sets and symbols can be provided by Unicode to describe content in non English languages. An interoperable XML enables information exchanges between individual organisational environments and highly diverse enterprise applications. For example, a transaction in XML format created on a Windows platform with an SQL Server 2000 database could be handled on a UNIX platform with an Oracle database, regarding both platforms and environments support XML and Unicode, which is actually a type of enterprise integration.

As mentioned earlier, a common form of integration is sharing, exchanging or moving files between legacy enterprise systems that were built a long time ago in different programming languages, on different platforms and database management systems. For example, account information maintained by an online shopping system might be shared and exchanged with the compliance management system and also with the payment

entry system. Each of these systems could be built on different hardware with different operating systems and databases. XML enables data exchange between them as long as they support the processing of Unicode character encoding and have their own XML parsers.

XML has become one of the most important technologies for interoperable and collaborative computing. In addition to new systems with built in XML support, extended XML support and capabilities are being provided by most of the major technology vendors. While XML-based applications are diverse, XML parsers, software development kits and database extensions for data import, export and persistence in XML format have been standardised.

For the implementation of a software evolution tool, one of the challenges is to tackle numerous file formats, different programming languages and platforms, which is particular significant when facing Web-based systems. As more and more XML extension tools are developed for various computing environments, an XML transformation tool could define portable models for working with other XML-based tools to handle the heterogeneous computing environment. One of the issues addressed by this research is to define a data model that can be exported using an XML described file format and imported into other similar tools.

9.1.5 Flexibility

Static structures are often used to define database architectures and application interface files, where the attributes and sequence of information can not be easily changed without affecting dependent applications. In the previous example, interface files that contain a UK address are defined with two street address lines, city, county and post code. The structure will be unable to support or describe address formats of other countries. To represent address data with a static interface file for other countries, one of the following approaches can be taken:

- Use existing interface file.

- Extend existing interface files to support new addresses.

- Keep equivalent data and ignore other data.

- Develop new interface files.

In addition to the interface file, the source and target database architectures would need modification too. The first approach will cause confusion in understanding the metadata and thus interpreting the contained data correctly. The second approach means adding another type of address will again incur modification to the interface file. The third approach could lose important data and is least likely adopted.

Developing a new interface will solve this issue only if it is flexible enough to fit potential new address formats, which can be achieved by XML with structures engineered according to vertical models and architectural forms. A vertical model utilises groups of similar elements, while a flexible architectural form allows for the addition of new attributes and elements to an XML document, such as the insertion of additional address lines with minimal or no impact to dependent applications.

9.1.6 Reusability

One of the objectives for building evolvable Web-based systems is to reduce cost for development, maintenance and enhancement. Reusability can be achieved in two ways: design with the intent to reuse, and identify reusable component from existing system. Reuse is a good way to reduce cost only if it is carefully adopted. Reckless reuse could lead to error-prone, unstable systems. Even with good design, developers often avoid formalised reuse for the difficulty in applying practical reuse in development.

XML Schemas can facilitate reuse in numerous ways. As a metadata technology, it can be engineered as modular component schemas and structured to exhibit and support several types of reuse. Data types can be defined and reused via reference. In Figure 9-1-1, a custom data type is defined as "simpleType" to represent the enterprise standard for data type money and is reused by references.

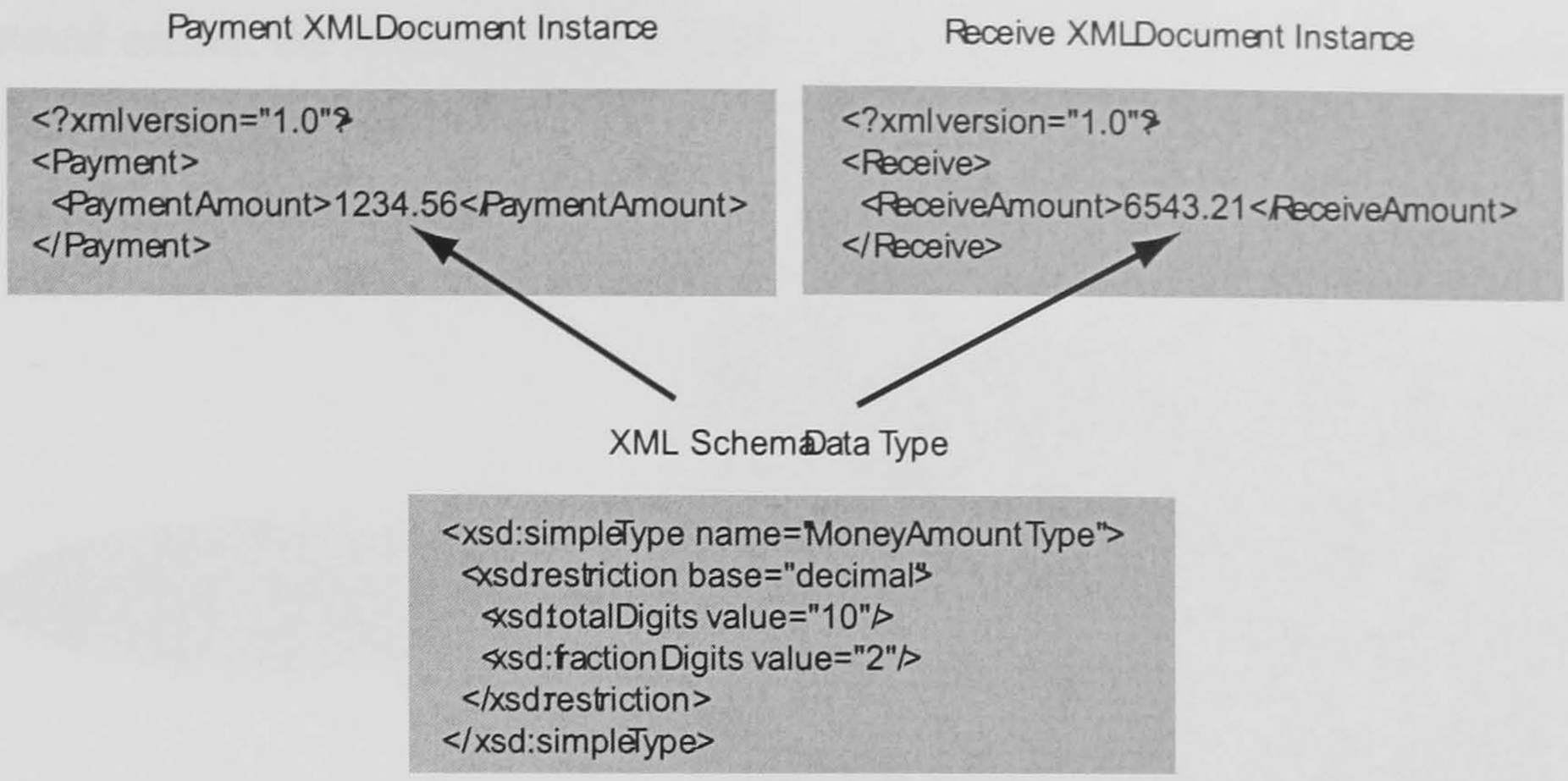

Figure 9-1-1 Data Reuse for Payment Amount.

In addition to reusable data types, XML Schemas allow modular external component via inclusion and reference between them. For example, an enterprise metadata standard for customer address required to support highly diverse postal address formats could be specified in an XML Schema, which serves as a standard and can be reused in various contexts. Future modifications for address structure can be made to the component address schema rather than all of the referencing schemas.

Standard schemas defined and published to an enterprise can reduce development and maintenance costs, where application-specific XML schemas reuse the component schemas via reference to avoid redevelopment of the same structures used by other applications.

9.1.7 Extensibility

Two features of XML provide its extensibility.

- Any set of element and attributes can be defined in XML for a particular application. There is no constraint in XML vocabulary other than simple rules for format definitions.

- A vocabulary can be defined as a base standard that all dependent applications should comply with. A specific participant can then extend the base standard without having impact on others.

As stated earlier, the most challenging task in Web-based systems evolution is constant changes in business paradigms, technology platforms and development practices. The extensibility of XML can facilitate a software system to accommodate changes.

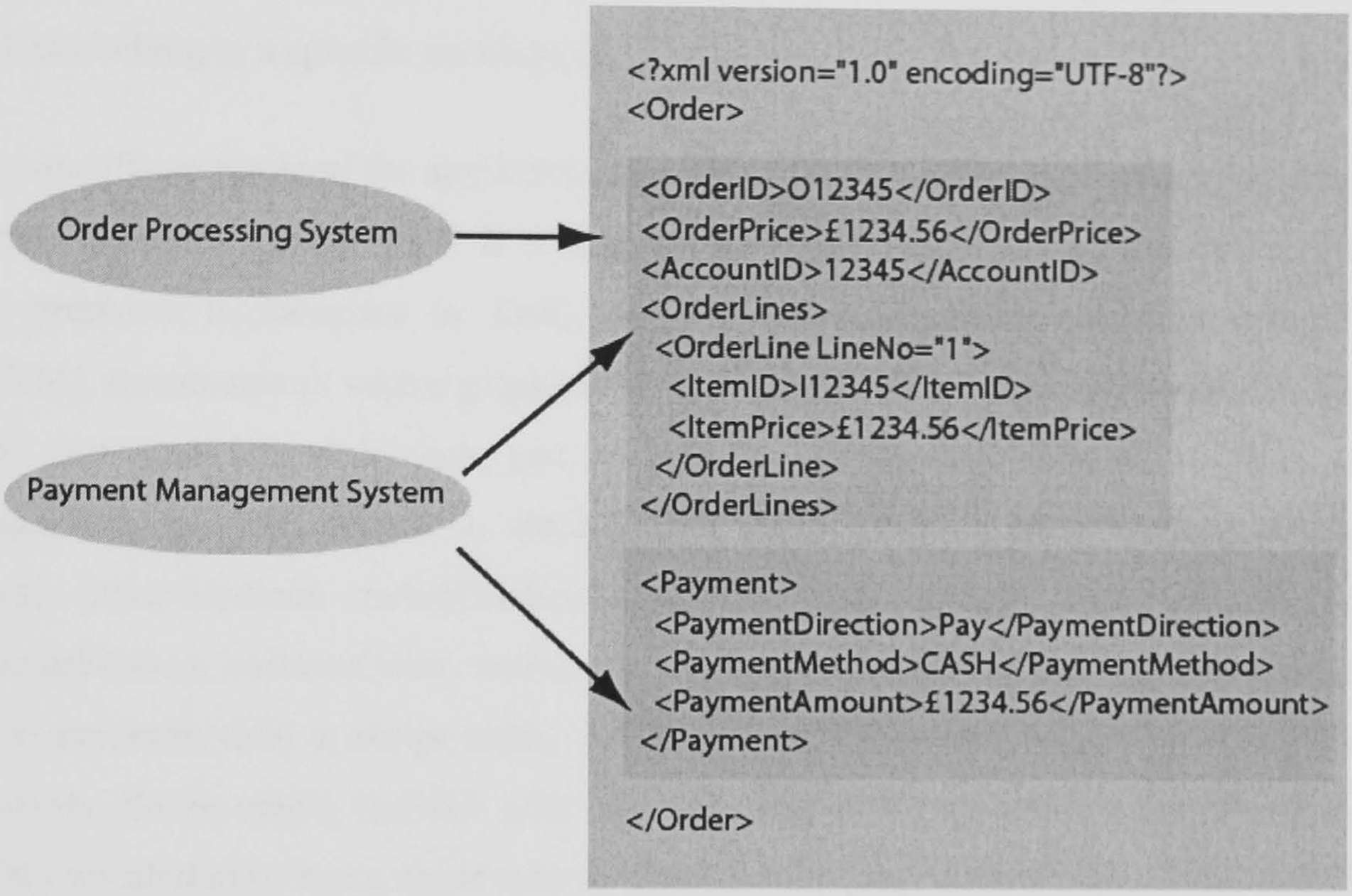

Figure 9-1-2 Extended XML Transaction File

For example, an online shopping system that provides products to consumers will receive cash payments via a broker. It needs to process order information but will leave payment handling to the broker. If a new payment method such as credit card needs to be offered to customers, changes have to be made to the order transaction files by adding new fields for credit card related information. An XML-based transaction file could be extended by adding new elements and attributes into its base XML transaction and schema or by referencing a separate component schema with the required information that would be processed by payment processing application but ignored by others. This is demonstrated in Figure 9-1-2.

In XML, a schema with a comprehensive set of data elements supporting a particular function becomes a vocabulary. An enterprise system will have a number of vocabularies for different functions. XML Schemas can be organised via unique qualifiers or namespaces, which allow adding new schemas or vocabularies with

minimal impact on the original XML schema. A namespace uniquely identifies a schema to avoid conflicts with others having similar elements and attributes. It is usually specified as a resource identifier such as a universal resource identifier or URI. A prefix is used as an abbreviation to a namespace. Elements or attributes with such prefixes belong to a specific namespace.

One specific example of the application of namespace in Web-based systems is Scalable Vector Graphics (SVG). SVG is a language for describing two-dimensional graphics and graphical applications in XML. SVG documents are embedded in HTML or XHTML documents as vector graphics for Web pages. SVG elements include [W3C03] desc, title, metadata, defs, path, text, rect, circle, ellipse, line, polyline, polygon, use, image, svg, g, view, switch, a, altGlyphDef, script, style, symbol, marker, clipPath, mask, linearGradient, radialGradient, pattern, filter, cursor, font, animate, set, animateMotion, animateColor, animate-Transform, color-profile and font-face. Five of these elements, title, a, script, style, and font, have the same names with five HTML elements. Some others conflict with MathML and Resource Description Framework (RDF) vocabularies. For a client side browser, a Web page with HTML, SVG and RDF content needs to be able to identify the exact type of any given element name that exists in more than one vocabulary.

Namespaces allow the elements or attributes with same names in different vocabularies to be associated with a unique qualifier. All three vocabularies have their own namespaces. Any element or attribute in a Web page belongs to one and only one namespace. This applies to all industry vocabularies specified via XML Schemas. Conflicts between elements or attributes with same names would be avoided to maintain the integrity of all individual vocabularies.

9.1.8 Hierarchical Composition

Compared to flat file formats or database record sets, the hierarchical structure of XML provides a rich format for representing complex information packaged in a flexible and standard way. Hierarchical structure is an inherent nature of most business information. A business typically keeps a diversity of information, where one data unit could have

nested data or could be nested in other data unit. This hierarchical structure can be mapped on to XML file with nested levels representing various types of information.

One of the benefits using hierarchical structure is the reduction of requests to a server, where a single request can retrieve an entire set of information contained in an XML file. For traditional methods accessing relational database, one request for top-level information and successive queries for more detailed information have to be made to retrieve the whole information. To reduce requests without using hierarchical structure, functionality for complex server-side data manipulation has to be built.

9.1.9 Internationalisation

Using Unicode not only provides XML the interoperability, but the means for internationalisation. XML uses ISO-10646/Unicode for the logical content of an XML document, where the content of any element and attribute can be Unicode characters representing various supported languages around the world.

In addition to various languages support, XML is designed to include mechanisms to identify languages and encodings, which allows building applications that can be used in many geographic regions. Web-based systems could be localised, internationalised or globalised systems. The different level of application scope requires many contributory factors, one of which is a data format that fully supports internationalisation. XML is best suited for this requirement and servers as the foundation for the internationalisation of Web-based systems.

9.1.10 Supporting Tools

Parsing and validation are two underpinning techniques for building XML-based applications. Software development practices can be greatly facilitated by using XML format and standard parsing and validation tools, where XML documents are specified in agreed format and checked against their specifications. The parsing and validation are performed automatically, and XML APIs such as the Document Object Model (DOM) and Simple API for XML (SAX) are used as XML navigation mechanism to access the contained information.

Parsing and validation tools are freely available for nearly all kinds of platforms. With the standard parser and validation tools, developers do not have to design, implement and maintain their own parsing and validating mechanisms. In addition, XML documents created by one application can be parsed and validated by others. The development and maintenance of highly heterogeneous environment of Web-based systems can greatly benefit from such features of XML.

9.1.11 Highly Adapted for Web-based Systems

One of the major objectives in the design of XML was to create a data representation standard compatible with the Web and Internet. Such compatibility is established at the level of Hypertext Transport Protocol (HTTP), over which XML documents can move between client browsers and servers just like HTML. XML applications can make use of the many related standards built upon HTTP, such as authentication and encryption. In addition to HTTP, XML is supported by many other Web related protocols such as FTP, e-mail and message queues. Any mechanism capable of moving data can be used to transmit XML documents.

The high degree of acceptance and implementation of XML makes it a de facto standard for achieving data management in Web-based systems. All emerging Web-related technologies are built on XML such as Web-services, Web portal and Grid computing.

9.2 XML Integration

The integration patterns introduced in the previous chapter can be implemented via XML in various forms. The features of XML make it a natural choice for integration which is all about transferring information between applications, databases and systems. XML can be seamlessly combined with existing integration infrastructure. Its standard format and tool support will greatly reduce the time and cost involved in integration project.

9.2.1 XML-based APIs

Third party software packages often have their functionality exposed via XML-based application programming interfaces (API), which allow enterprise applications to integrate with the third party application. For example, Microsoft SQL Server 2000

Supports XML queries and results, XML access via HTTP and updating SQL Server via XML documents so that SQL Server and related applications can access XML documents like any other type of SQL Server data.

9.2.2 Aggregation

XML provides a mechanism for an organisation to establish logical views of enterprise information. Business objects that dominate the enterprise application can be represented via the hierarchical structure of XML. Complex structures such as client can be specified in XML Schemas which define data structures populated from one or more data sources of the enterprise. The use of XML and XML schema create a layer of abstraction for enterprise applications, where business objects can be manipulated by applications without knowing the details of the data sources.

9.2.3 Business to Business Integration

Business to Business Integration aims to automate business processes between companies, where one company markets and provides goods and services to another and both participants of a B2B e-commerce model may also be collaborators or trading partners under a formal or informal contract of agreed-upon terms such as scheduling, pricing, delivery, and support. The process automation is achieved by performing these functions electronically via messages such as Electronic Data Interchange (EDI).

While EDI is expensive to implement and is dominated by a few industry giants, the features of XML make it a natural candidate for replacing EDI. Specifications such as ebXML have been established for electronic business framework, which enables a global electronic marketplace for enterprises of any size and in any geographical location to meet and conduct business with each other through the exchange of XML-based messages. XML has played a key role in removing barriers to electronic business by providing an inexpensive infrastructure to all participants.

9.2.4 Information Exchange and Distribution

Information exchange occurs in industries and government agencies, which need to exchange information for reporting, monitoring and management. XML is the de facto format for exchanging such information and has been used as the core technology in

electronic government initiatives for information exchange. A common scenario among both government and industry groups is the sharing of information, where a central authority creates an XML Schema or format that can be used by participants to move information in an agreed format.

In addition to information exchange, XML is adopted as the main technology in information distribution mechanisms such as RSS (Rich Site Summary, or Really Simple Syndication for RSS 2.0), an XML format for content publication of Web site. RSS is a simple XML format used for content syndication by numerous web sites ranging from personal web logs to major newspapers to government agencies, where a continuing feed of new information is provided to interested readers. RSS is used in a wide range of applications such as web logs, software updates, security bulletins, government regulations, court decisions, art gallery openings and office calendars. RSS is a typical application of information distribution via the platform and application independence of XML.

9.2.5 Web Services

Web services is an implementation of service-based pattern. It builds a distributed programming model on XML standards, where a service consumer initiates an operation of a service specified in XML via XML-based SOAP standard. XML is the key technology that enables a large number of participants to corporate in a distributed programming environment.

Web services is one of the most important technologies for enterprise legacy systems integration, where it could be adopted in three ways depending on the exposed services. Figure 9-2-1 shows different Web services integration strategies for legacy data, legacy API/method and contract.

Exposed Services	Service Sources	Approach
Legacy Data	Legacy data such as database tables, files and messages	Develop XML Schema for legacy data and SOAP as the message format
Legacy API/Method	Legacy remote methods or program APIs	Define XML data types for the API/method parameters and SOAP as the message format

Contract	Contract derived from shared business data	Define interfaces of the contract and develop wrappers for legacy systems to provide contract implementation via legacy data, messages and APIs

Table 9-2-1 Web services Integration

Web services technologies allow legacy and new applications to interoperate. It can create composite applications by combining interfaces to individual applications with various data sources using XML as the standard data format.

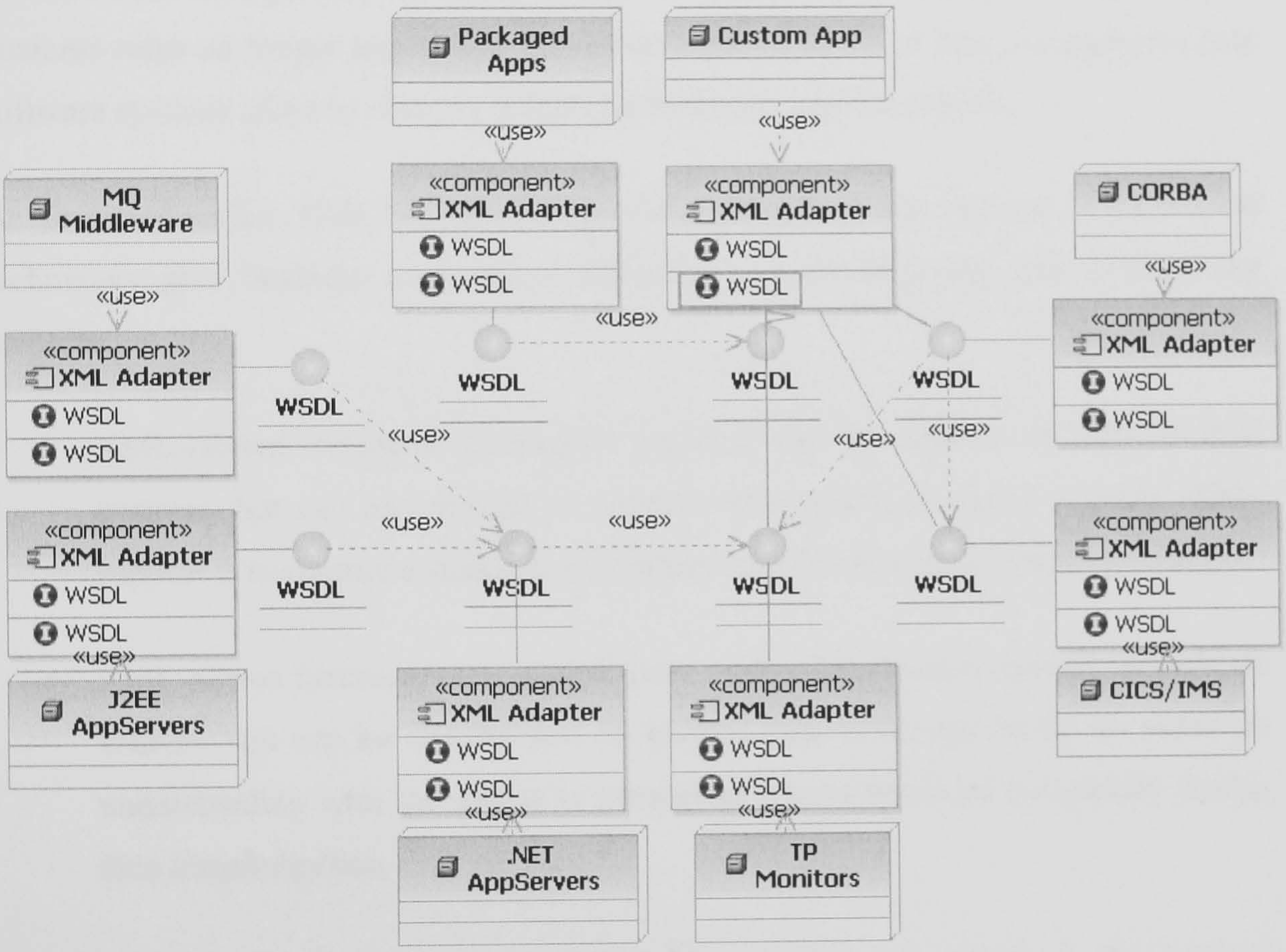

Figure 9-2-1 Web services Integration

In Figure 9-2-1, XML messages are passed in a standard format from one node to another. Each node is responsible for transforming its native data types and formats to and from XML messages to communicate with others.

XML in the proposed solution to Web-based Systems Evolution has two roles: model documentation and systems integration. This research aims to develop an integration solution based on the technologies of patterns and XML data management introduced in previous sections and chapters. Model documentation in XML is given in Chapter 2.

9.3 Benefits of Using XML in Software Evolution

One of the challenges software evolution faces is handling changes from business requirements and technology development. A typical scenario is that a business changes its processes to enable information distribution to clients and suppliers via the Web and Internet. Such changes require incorporating new technologies with some of the original business rules no longer applicable. The advantages of XML in data management help software systems adapt to changes in both technologies and businesses.

As discussed earlier, XML has been integrated in all mainstream systems. XML related techniques give business competitive advantages in development and modernising projects.

- XML-related evolution techniques not only can be applied on XML-related systems, but can and should be adopted for traditional legacy systems. XML support is becoming a standard for systems capable of accommodating changes.

- XML allows heterogeneous connectivity, where information appears in multiple sources and can be transformed to various data structures based on semantic understanding with the ability to managing greater levels of complexity during data transformation.

- XML is the de facto standard for data management, which is the core of enterprise applications. Software reengineering can take advantage of XML, not just at the technical systems level, but at the business level as well.

- Heterogeneous legacy systems enhanced with XML transformations can perform functions with the systems cooperating as a result of integration.

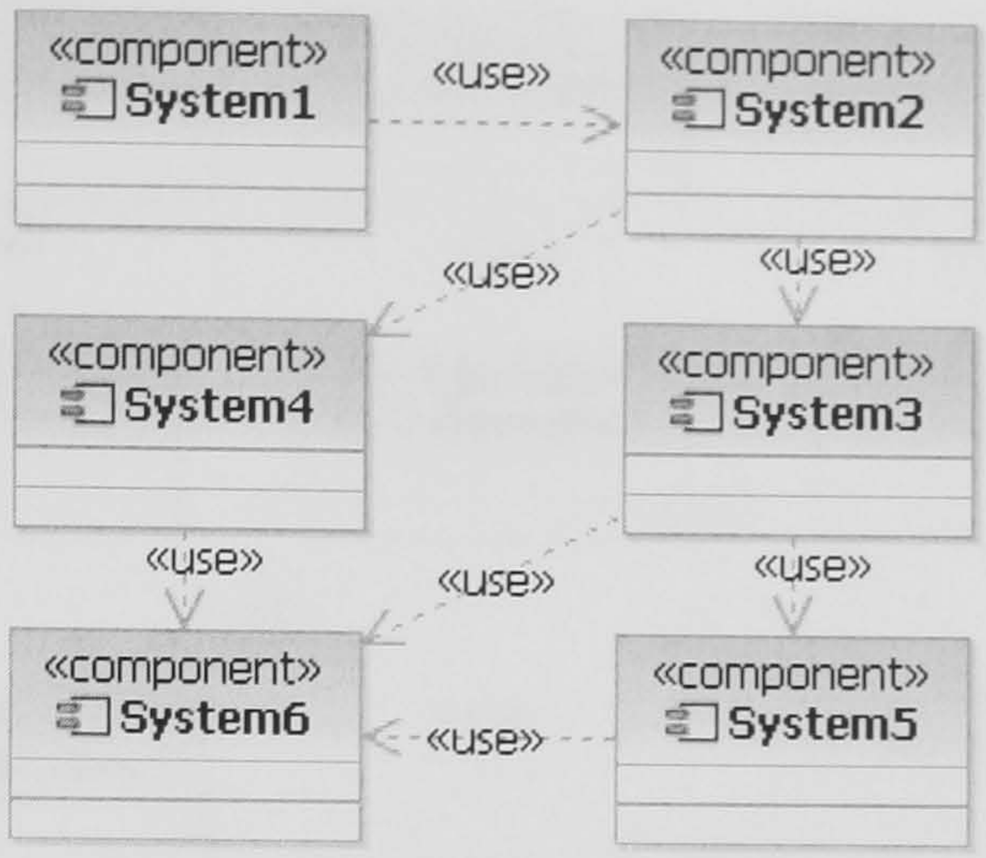

Figure 9-3-1 System flow chart before reengineering

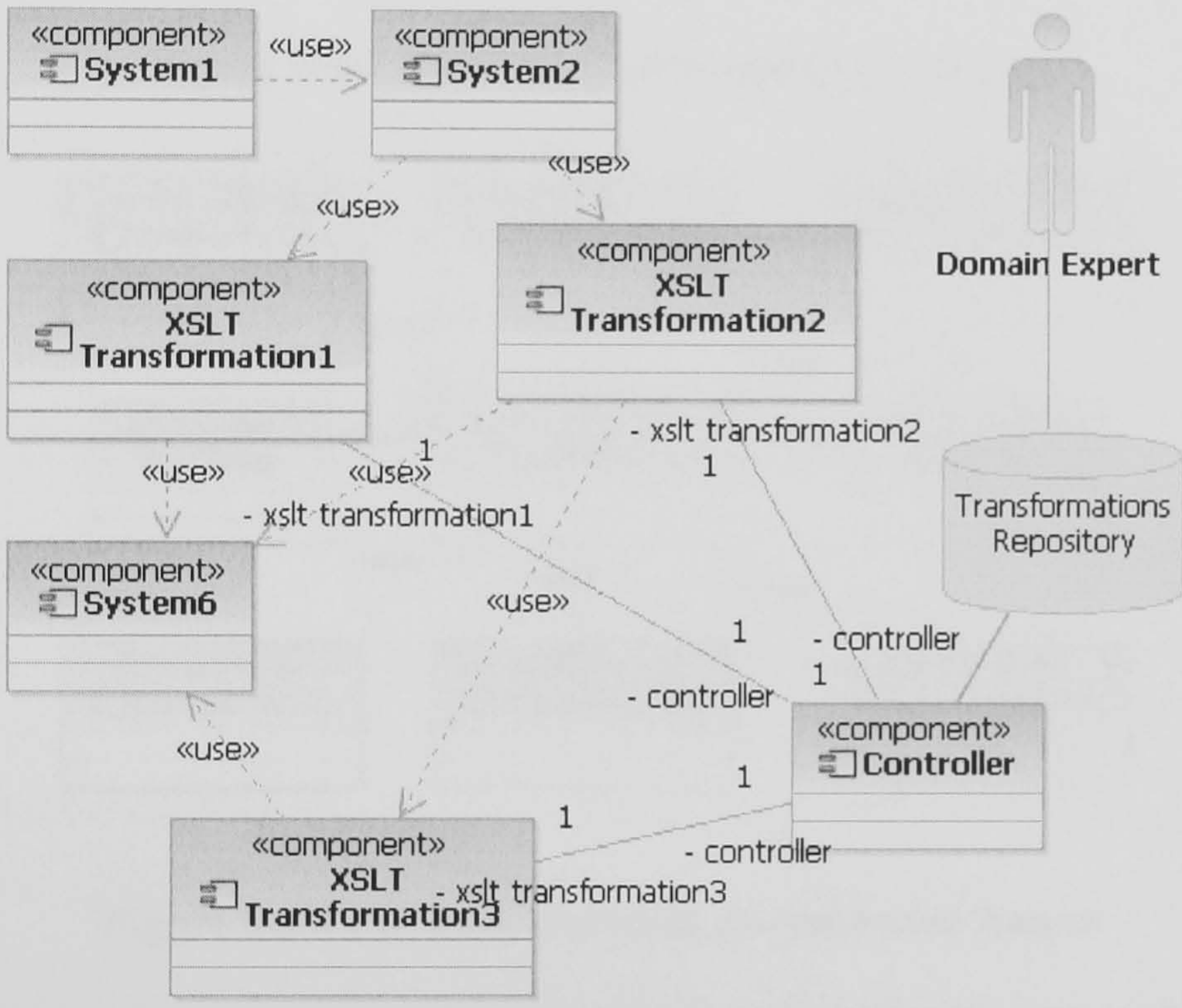

Figure 9-3-2 XSLT Transformation for System Evolution

XML should be a foundation for all evolution activities such as specifying the problem domain and its relationship with business terminology. Figure 9-3-1 and 9-3-2 show a transformation replacement to Systems 3, 4 and 5 based on XML provides better control over the business process by reflecting the domain knowledge.

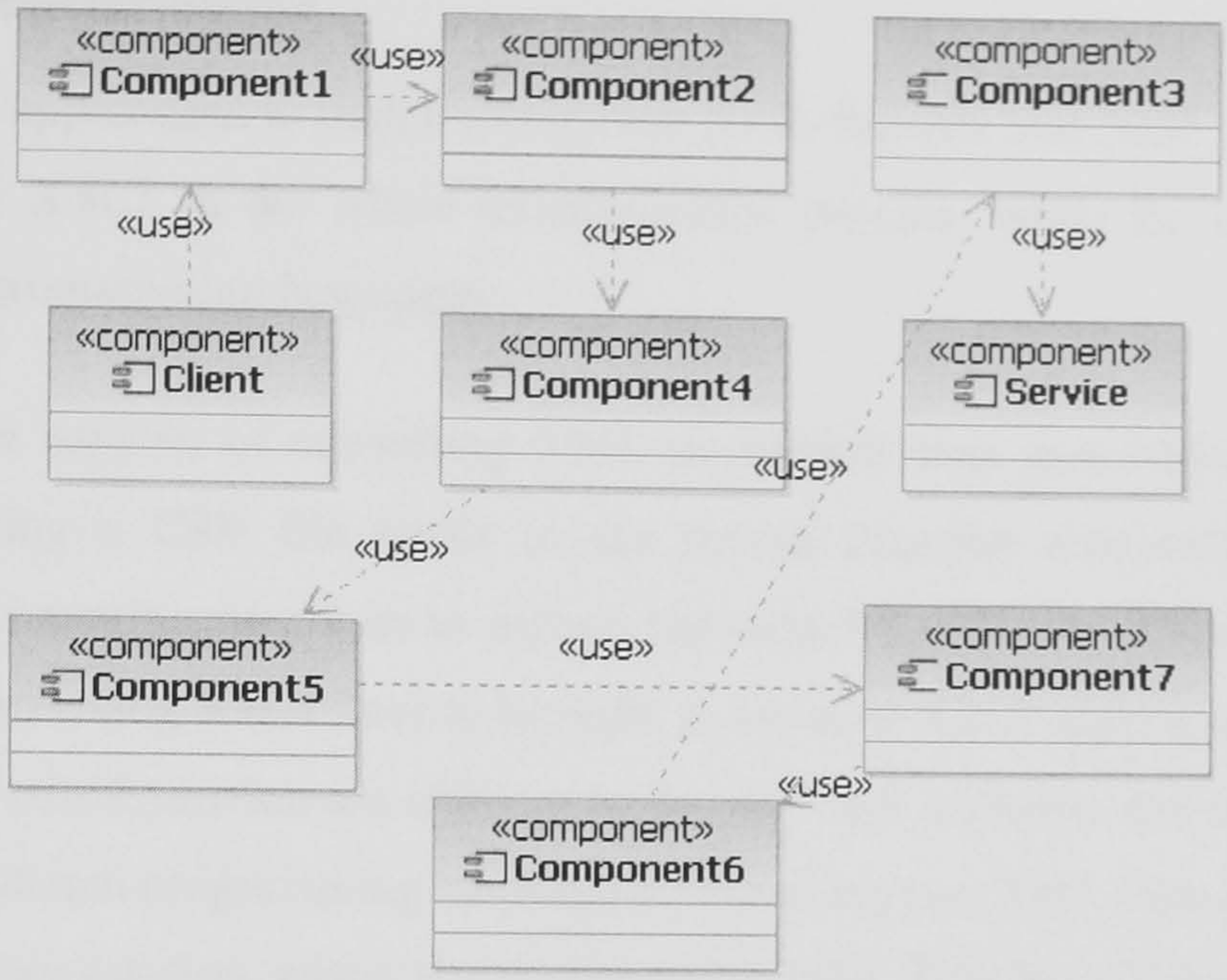

Figure 9-3-3 Surf-and-seek Portal-based Access

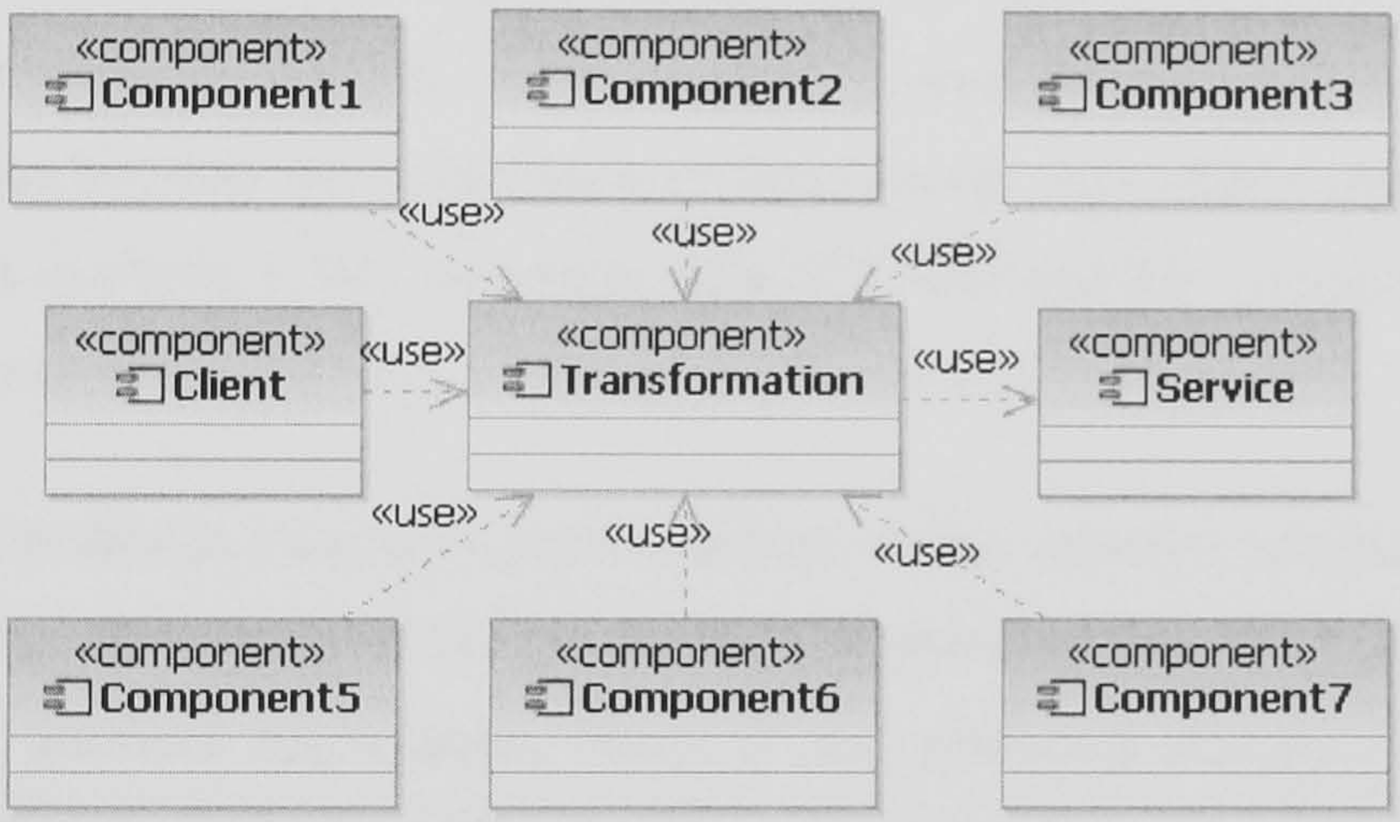

Figure 9-3-4 Focused, Habitual, Portal-based Access

XML based portals can be used to increase business throughput by accomplishing more tasks with less components of the infrastructure. Figure 9-3-3 and 9-3-4 [Lanham01] show how the XML-based portal can dramatically affect the information flow and access within organisations.

The task of XML-aware evolution is to perform various types of data format conversions for systems integration. The design issues for XML-aware evolution are discussed in the following sections.

9.4 XML Transformation

There are two approaches to transforming Non XML formats into XML. One approach is to use only XSLT in the whole transformation process, while the other is to use conventional programming languages.

XSLT is more capable of converting XML documents than non XML formats. For example, parsing a CSV file needs to use search function repeatedly to find the delimiter and substring functions to extract individual fields. For different non XML formats, corresponding XSLT have to be built. A series of XSLT stylesheets for specific file structures transformation are difficult to develop and maintain. On the other hand, using a conventional programming language for handling non-XML files can implement a general purpose solution, where simple parameter XML files handle the differences in file structures.

XML should be the common representation for any data sources to be exchanged between heterogeneous systems. Such common representation implies not only the common format but also the compliance of data content with native schema language data types. For example, a date compliant with ISO 8601 and the schema date data type should be represented as 2005-01-10 instead of 10-Jan-2005 or 01/10/05.

It is important to design a common representation of data content compliant with native schema language data types, which are the data representations to be used natively by XML related business applications. Most of the emerging standards for common business documents, such as purchase orders and invoices, are established on these built-in types. Using these representations can avoid writing the same convention code into XSLT stylesheets.

The analysis of grammars of non XML formats requires techniques from compiler construction and the formal study of languages. Each of the formats to be converted corresponds to a separate class of grammars. For example, the X12 EDI syntax is defined in the X12.6 standard.

The grammars involved in the Non XML formats transformation belong to a special class of grammars called "context-free grammars", which can be described via Backus-

Naur form (BNF). Among many variations of BNF, Extended Backus-Naur Form (EBNF) is specified by W3C as the XML 1.0 Recommendation and will be used as the notation in this research. EBNF grammars will be encoded in XML by creating an XML-based language for grammars description.

9.5 Design and Grammar

There are four components related to XML representation for grammars of legacy file formats: XML instance documents, XML grammar description documents and their corresponding XML schemas.

9.5.1 Instance Document Design

Principles adopted in instance document design are as follows.

- Only XML Elements are used for instance documents. Data entries in Legacy file structures do not dictate what should be an Attribute instead of an Element and using both Attributes and Elements complicates the transformation design.

- The structure of instance document design needs to match the logical hierarchy of the corresponding legacy format.

- Only unnamed default target namespace is used in instance documents. Named namespaces aim to distinguish Element and Attribute names. The XML instance documents, as intermediate formats for XSLT transformations, will not be used directly by other applications and thus there will be no conflict of duplicate names.

9.5.2 Grammar Description Document Design

In addition to specifying the grammars of legacy non-XML formats, the grammar description documents also specify characteristics of the input files, such as record terminators, and the output XML documents, such as schema URLs.

- The structure of grammar description documents correspond to a logical structure of the legacy file and grammar characteristics.

- Like instance document, only unnamed default target namespace is used in grammar description documents.

- Unlike instance documents, Elements are used for describing structure and Attributes for describing values in grammar description documents, where certain field characteristics could be more easily accessed as Attributes.

Element	Description
Grammar Element	Represent the root element of the XML grammar description document.
Group Element	Represent a group of records (Record Element) or sub-groups (Group Element).
Record Element	Represent the structure and characteristics of a record with a group of fields (Field Element).
Field Element	Represent a field in legacy format with a group of attributes for characteristics such as name, data type and number.

Table 9-5-1 Elements of Grammar Description Document

The grammars of legacy formats are constructed via basic elements shown in Table 9-5-1.

9.5.3 Schemas for Grammar Description Documents

Grammar description documents are validated against their schemas. Some issues need to be considered as follows.

- Named types will be suffixed with "Type".

- One grammar schema is specified for transformations with legacy format as the source; another is specified for transformations with the legacy format as the target. Similar named types in the two schemas will be declared in a common schema for a specific legacy format, which is included in both the source and target schemas. A global schema is specified for types referenced in all the transformations. Figure 9-5-1 shows the hierarchy of the grammar schemas.

- In stead of global Elements, named types and local Elements are used to specify grammar schemas, which are analogous to the hierarchy of class and subclass.

- In the common schemas, reusable types are named types. In instance document schemas, the root Element declaration uses an in-line anonymous type declaration to hold its child Elements as a sequence.

- As discussed earlier, named namespaces will not be used for the sake of simplicity.

9.5.4 Schemas for Instance Documents

Schemas play an important role in enforcing business constraints by validating XML formats against schemas that specify the XML representations of non-XML formats. XML APIs provide the mechanism for schema validation.

A one-for-one correspondence is created between the XML and the non-XML formats, where syntax transformation of data complying with business constraints defined in schemas between an XML representation and a non-XML representation will not result in different semantics.

9.6 Architecture of XML-aware Evolution

The XML transformation process is divided into two categories according to whether the legacy format is the source or target format. In either case, the transformations share a common structure. This section discusses the design of XML-aware evolution. Chapter 10 introduces the implementation details.

9.6.1 Source Transformer Processing

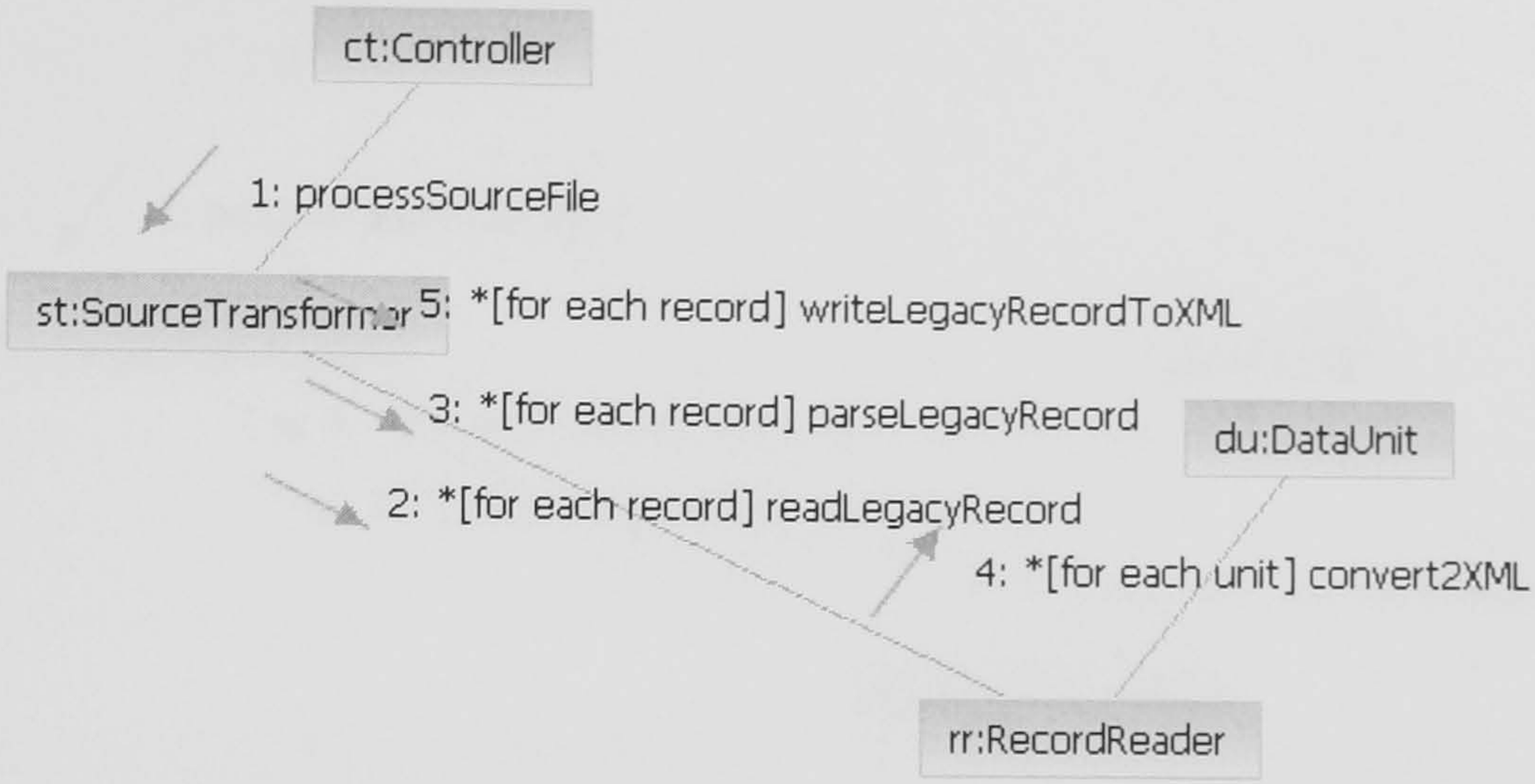

Figure 9-6-1 Source Transformer Collaboration Diagram

Figure 9-6-1 shows a collaboration diagram of the objects of source transformer. The controller takes arguments and invokes the processSourceFile method of a SourceTransformer object for a specific legacy format. The processSourceFile method creates directories for the output XML documents, creates and serialises DOM Document objects. For each record in the source file, the readLegacyRecord and parseLegacyRecord methods of the RecordReader object for a specific legacy format are invoked to analyse the individual units of a record, for each of which a DataUnit object is created with the corresponding class type. Next, the convert2XML method of each DataUnit object is invoked to transform the contents of the field from legacy data type to a corresponding schema language data type. Finally, the processSourceFile method invokes the writeLegacyRecordToXML method of RecordReader object to write the DataUnit objects for a legacy record to corresponding DOM Elements.

9.6.2 Target Transformer Processing

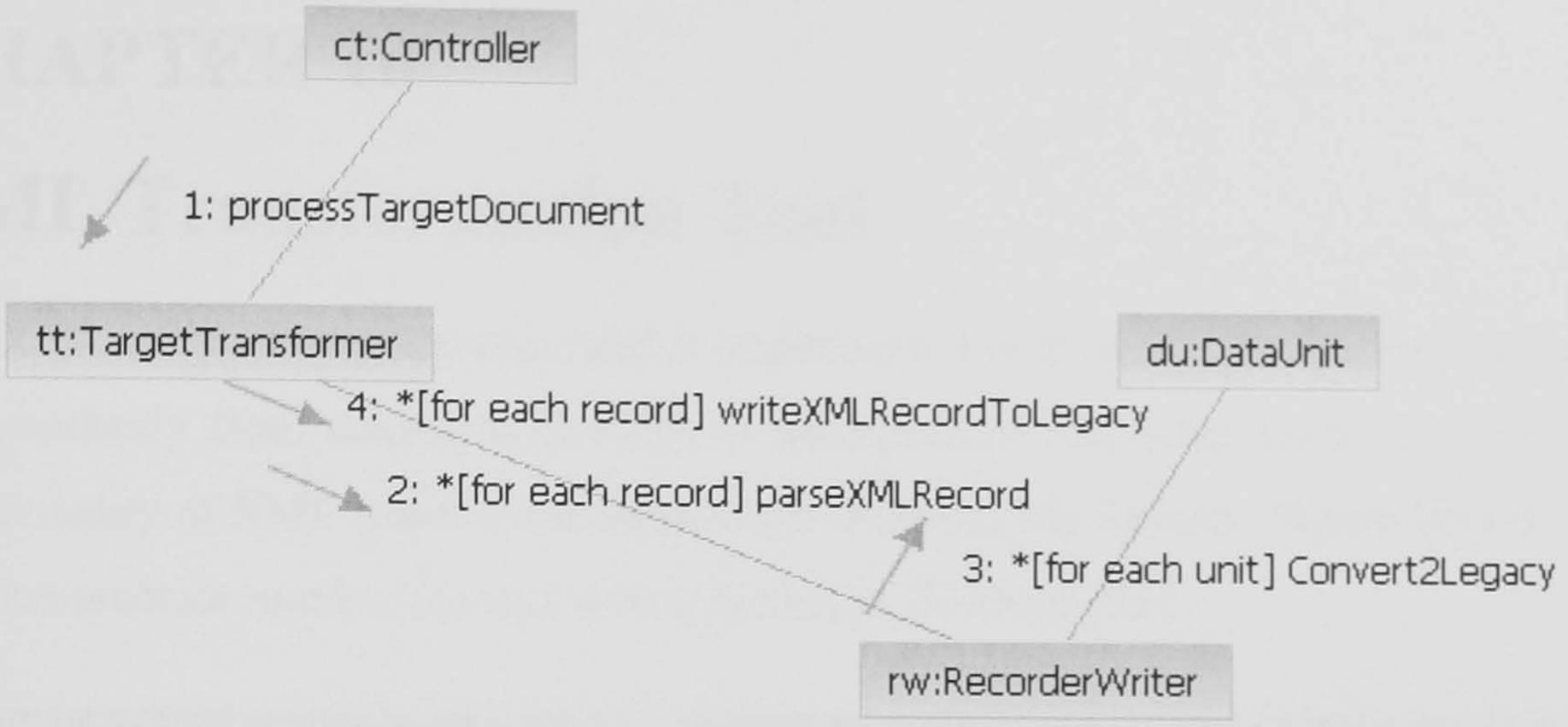

Figure 9-6-2 Target Transformer Collaboration Diagram

Figure 9-6-2 shows a collaboration diagram of the objects of target transformer. The controller routine takes arguments and reads the XML documents to process. For each XML document file in the input directory it invokes the processTargetFile method of TargetTransformer object to parse the input XML document file and load it into a DOM Document object tree. Next, the parseXMLRecord method of RecordWriter object is invoked for each Element representing a record in the legacy format to create DataUnit object for each Element representing a field in the legacy format record. The convert2Legacy method of each DataUnit object is invoked to transform the data representation from the schema language data type to the legacy format data type. Finally, the writeXMLRecordToLegacy of RecordWriter object is invoked to write the legacy format record.

9.7 Summary

This chapter analyses the roles of XML in data management for Web-based systems evolution. Section 1 explores the characteristics of XML-based data management. Section 2 introduces various XML-based integration strategies. Section 3 analyses the application of XML-based data management in software evolution. Sections 4, 5 and 6 discuss the design of an XML transformation tool.

CHAPTER 10

XML Transformation Tool

The XML-aware Transformation tool is implemented in Java as a jar file that can be run independently from console/IDE tools, or incorporated into other tools to provide the functionality of XML transformation between different file formats. Figure 10-1-1 ~ 10-1-3 demonstrate running the tool within Netbeans [Netbeans05].

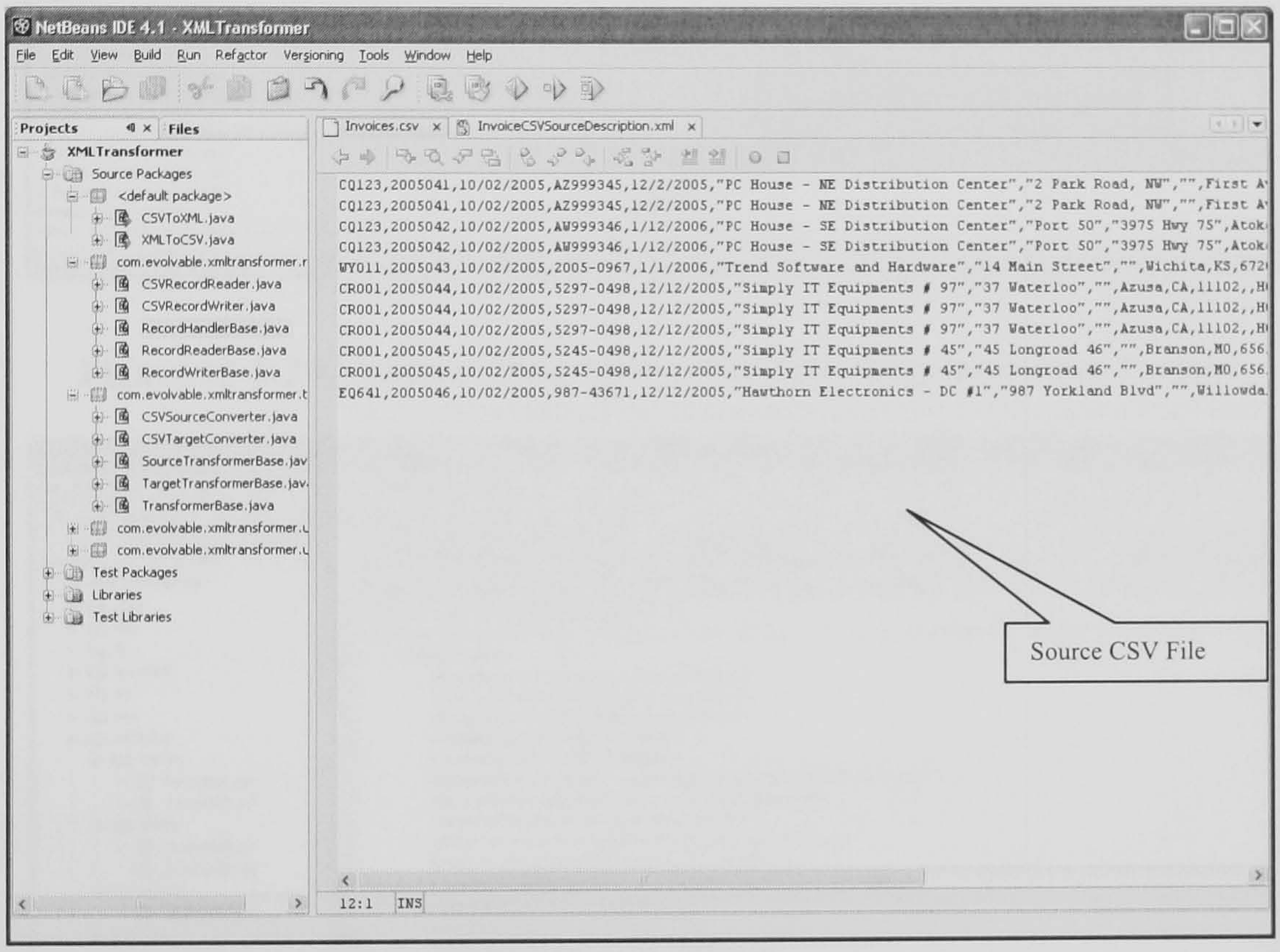

Figure 10-1-1 Source CSV File

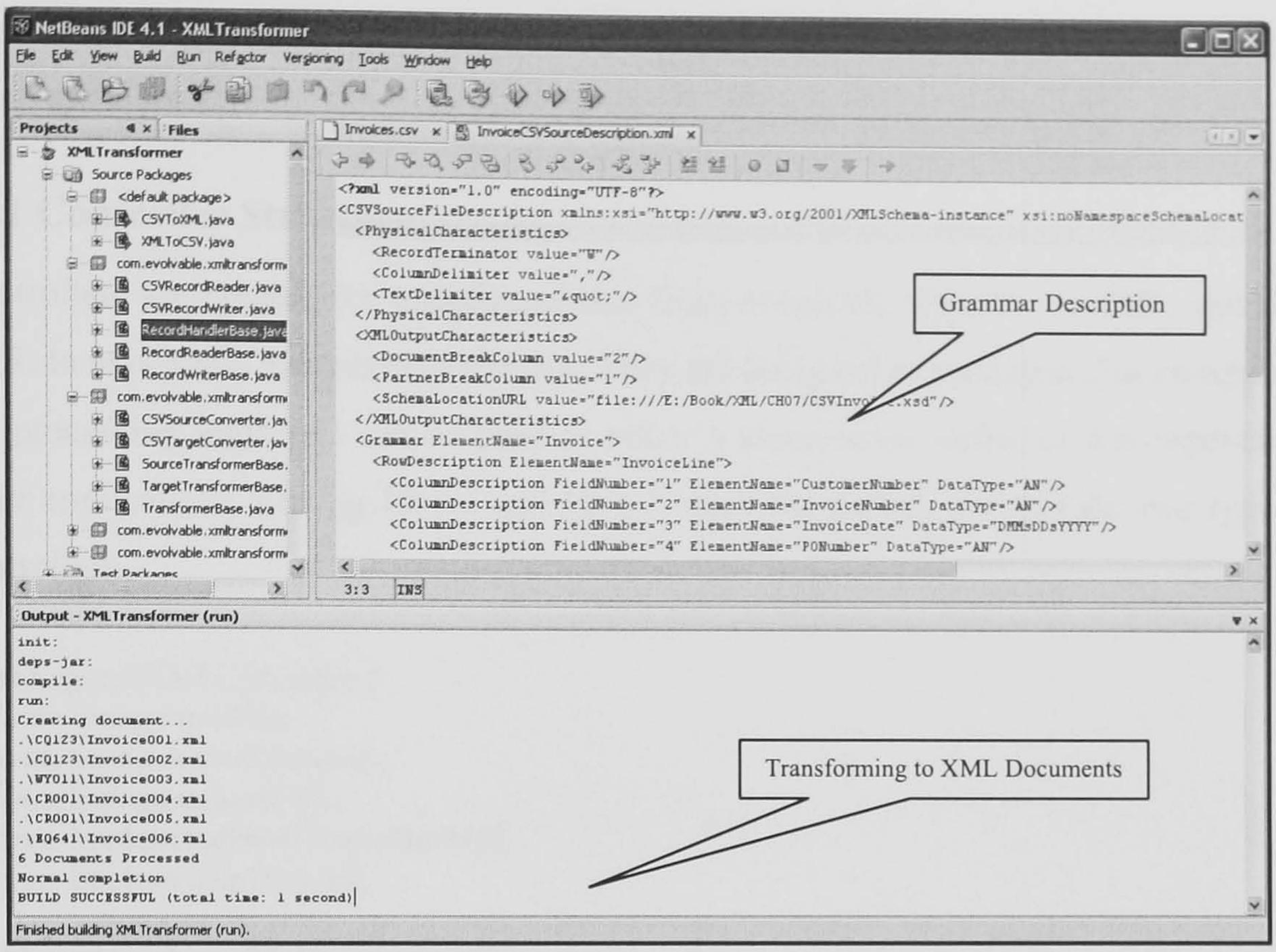

Figure 10-1-2 Grammar Description File and Transformation Process

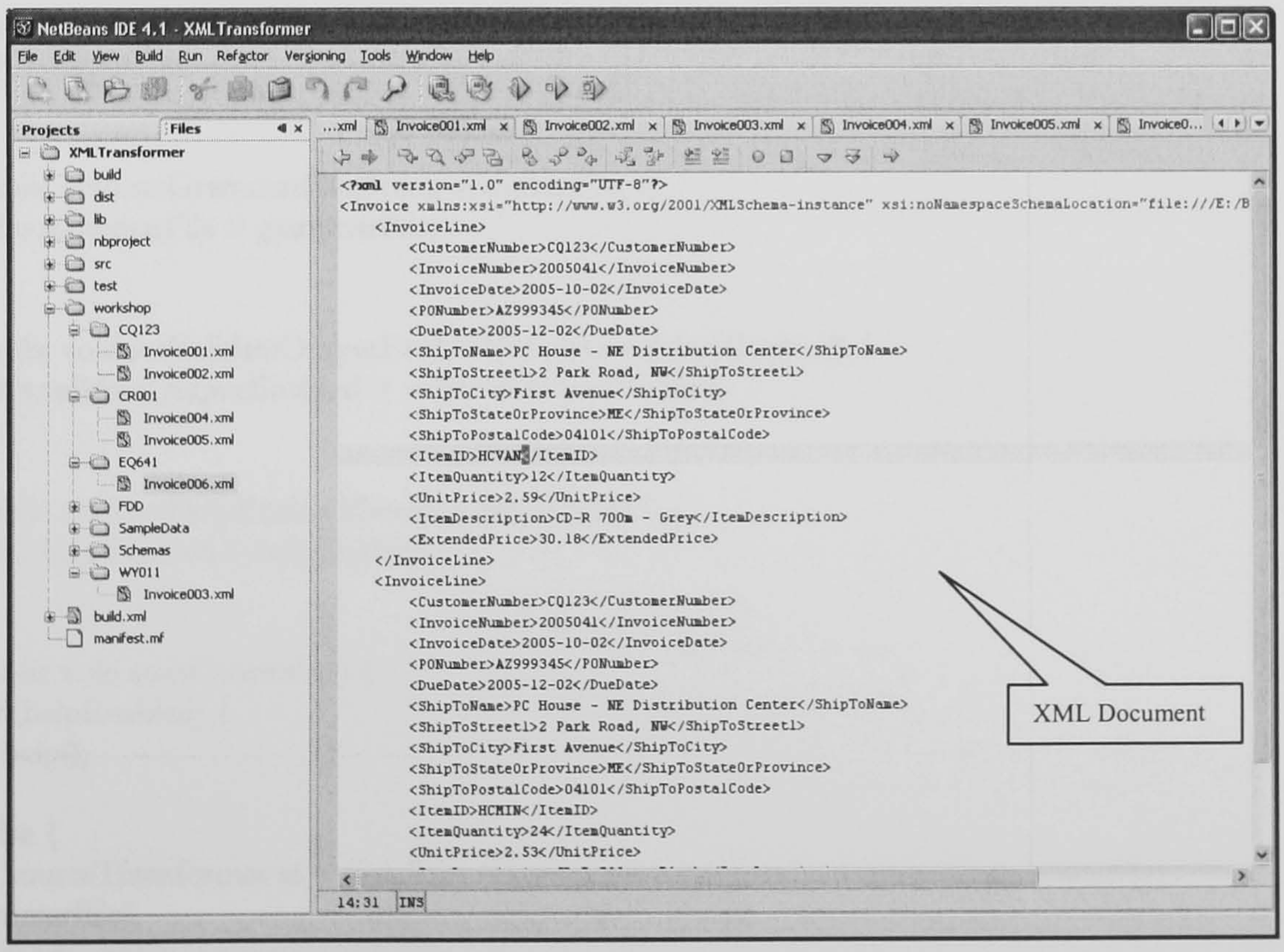

Figure 10-1-3 Resulting XML Documents

The controller routines and base classes for XML-aware evolution are explained in detail as follows.

10.1 Controller Structure

Controllers are created for transformations from non-XML formats to XML and from XML to non-XML formats respectively. They are designed as packages that encapsulate the processing logic and can be used as either a stand-alone utility or a component of other applications. Listing 10-1-1 and 10-1-2 show the pseudo code of the two types of controllers.

```
Class Legacy2XMLController {
  private String inputFile;
  private String outputDirectory;
  private String grammarFile;
  private boolean validateOutputEnabled;
  private boolean helpEnabled;

  public void setInputFile(String inputFile) {
    this.inputFile = inputFile;
  }

  public void setOutputDirectory(String outputDirectory) {
    this.outputDirectory = outputDirectory;
  }

public void setGrammarFile(String grammarFile) {
  this.grammarFile = grammarFile;
}

public void setValidateOutputEnabled(boolean validateEnabled) {
  this.validateOutputEnabled = validateOutputEnabled;
}

public void setHelpEnabled(boolean helpEnabled) {
  this.helpEnabled = helpEnabled;
}

public void startController() {
  if (helpEnabled) {
    help();
  }
    else {
      SourceTransformer st = new SourceTransformer(validateOutputEnabled, outputDirectory,
grammarFile);
      st.processSourceFile(inputFile);
      System.out.print("Transformation Completed!");
    }
  }
```

```
  public void help() {
    System.out.print("Help ...");
  }
}
```

Listing 10-1-1 Controller for Legacy to XML Transformation

```
Class XML2LegacyController {
  private String inputDirectory;
  private String outputFile;
  private String grammarFile;
  private boolean validateInputEnabled;
  private boolean helpEnabled;

  public void setoutputFile(String outputFile) {
    this.outputFile = outputFile;
  }

  public void setinputDirectory(String inputDirectory) {
    this.inputDirectory = inputDirectory;
  }

  public void setGrammarFile(String grammarFile) {
    this.grammarFile = grammarFile;
  }

  public void setValidateInputEnabled(boolean validateEnabled) {
    this.validateInputEnabled = validateInputEnabled;
  }

  public void setHelpEnabled(boolean helpEnabled) {
    this.helpEnabled = helpEnabled;
  }

  public void startController() {
    if (helpEnabled) {
      help();
    }
    else {
      TargetTransformer st = new TargetTransformer(validateInputEnabled, outputFile, grammarFile);
      setupDOMEnv();
      List<Document> docList = getAllDocuments(inputDirectory);
      while (Document d : docList) {
        boolean validate = false;
        if (validateInputEnabled)
          validate = validateDocument(d);
        if (validate)
          st.processTargetDocument(d);
      }
      System.out.print("Transformation Completed!");
    }
  }
```

```
  public void help() {
    System.out.print("Help ...");
  }
}
```

Listing 10-1-2 Controller for XML to Legacy Transformation

The difference between transformations to and from XML lies in the structure of controllers. For transforming to XML most of the work occurs in handling the input arguments and all documents are written to the output directory. For transforming from XML, a list of files in the input directory need to be loaded, validated and parsed. The resulting DOM document for each file is passed to the processTargetDocument method to produce an output file.

To implement the transformer, a TransformerBase class is developed with source and target transformer subclasses, which are required as base classes for each legacy format.

10.2 TransformerBase Class

The TransformerBase class is the base class from which the source and target transformers are derived. The attributes and methods are shown in Figure 10-2-1.

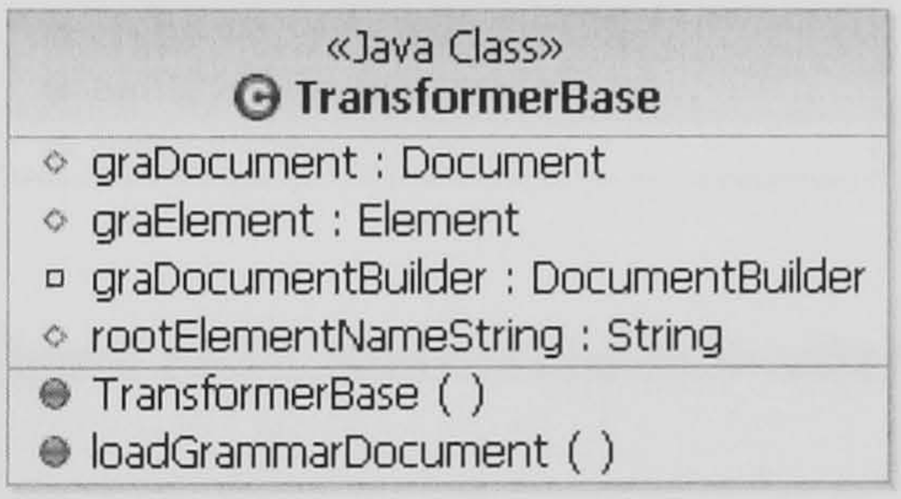

Figure 10-2-1 TransformerBase

10.2.1 Constructor

The constructor of base class TransformerBase sets up implementation dependent environment for loading and parsing the grammar document.

10.2.2 loadGrammarDocument

The loadGrammarDoccument method takes a grammar document name as parameter to load, parse and validate the grammar document. Next, it retrieves the grammar element

and root element name. As a separate method, it can be invoked many times for transformation of legacy formats such as EDI.

10.3 SourceTransformerBase Class

SourceTransformerBase is the base class for the all the legacy formats source transformer classes. Figure 10-3-1 shows the attributes and methods.

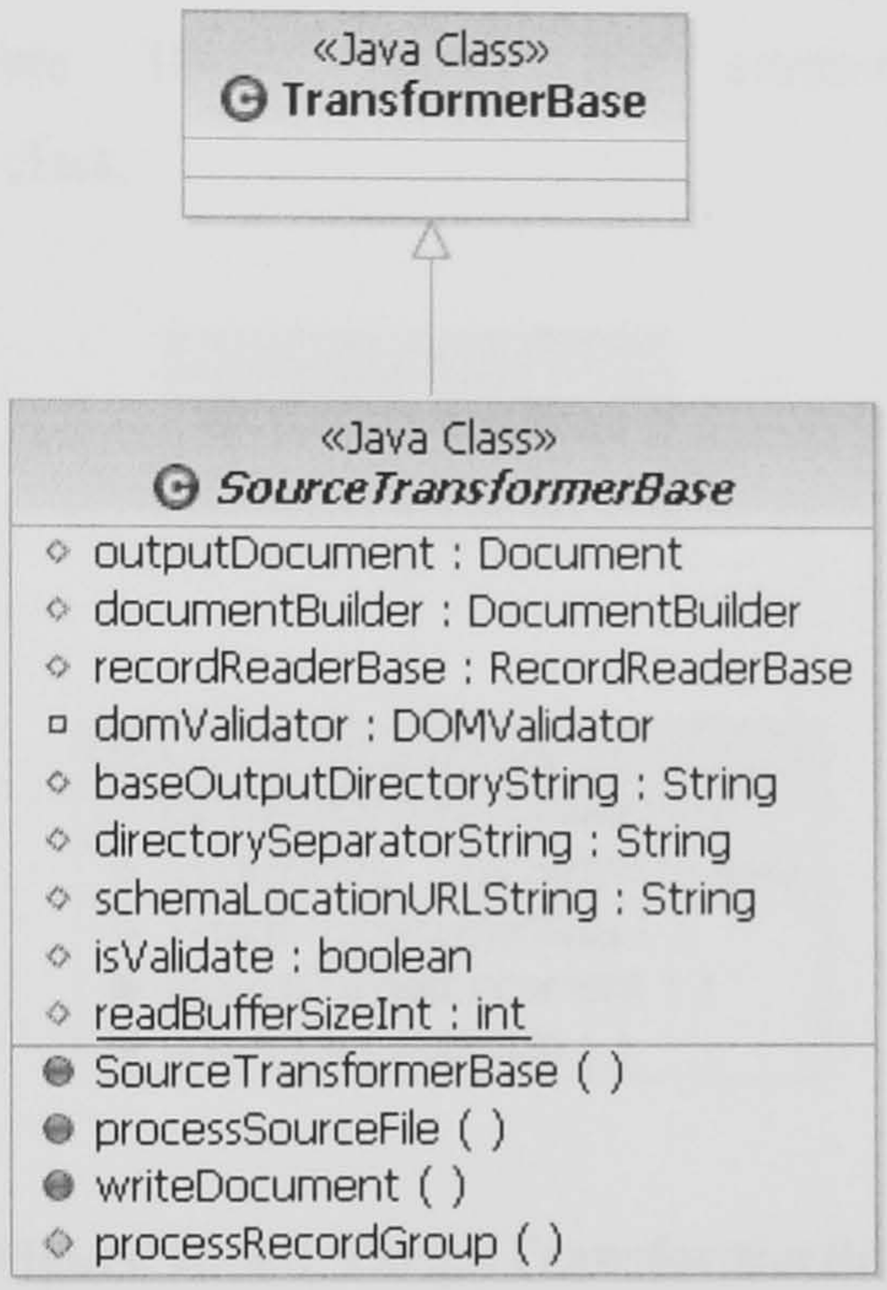

Figure 10-3-1 SourceTransformerBase

10.3.1 Constructor

The constructor of SourceTransformerBase takes the validation option and the name of base output directory as parameters to set the corresponding attributes. It sets up the document builder and DOM validation for DOM document output and validation.

10.3.2 writeDocument

The writeDocument method takes a document, a file name and the validation condition as parameters. If the validation is true, the output document is validated. Next, the document is written to an XML file with appropriate settings.

10.3.3 abstract processSourceFile

The processSourceFile method is an abstract method in the base class and is implemented in each of the subclasses for corresponding legacy formats. It takes the name of an input file as parameter and read the records of the file for processing.

10.4 TargetTransformerBase Class

The TargetTransformerBase class is the base class for all target transformers for various legacy formats. Figure 10-4-1 shows the attributes and methods of TargetTransformerBase class.

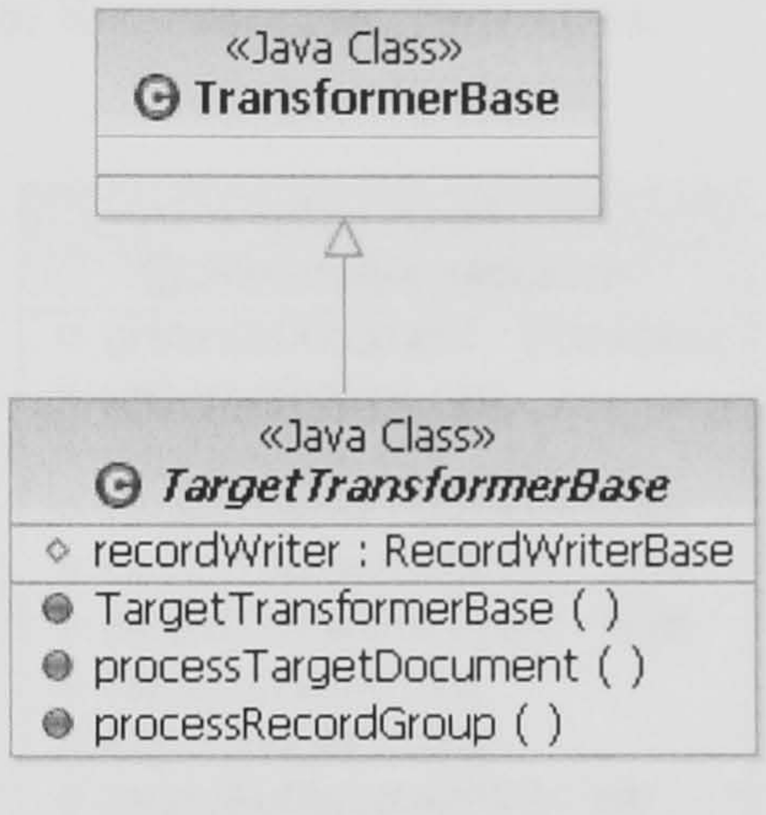

Figure 10-4-1 TargetTransformerBase

10.4.1 Constructor

The constructor of TargetTransformerBase class takes the name of grammar document as parameter. It invokes the base class constructor and loadGrammarDocument method. For source transformation, a single input legacy file may contain multiple documents that require loading multiple grammar documents for transformation. Such loadings will not be done in base class. For target transformation, a single input XML document will be transformed into one legacy file that requires only one grammar document. Such loading is done in base class.

10.4.2 abstract processTargetDocument

The processTargetDocument method is an abstract method in the base class that is implemented by derived subclasses for specific legacy formats. It takes an input XML

document as parameter to read from an XML file and create a single (or part of) output file in the legacy format. The input document is parsed against a grammar document and written to a file.

10.5 RecordHandlerBase Class

The RecordHandlerBase class is the base class of all record reader and writer classes, which handle records in legacy file formats. The two record terminators are used for to hold the record terminator character for variable length files and the segment terminator for EDI. The second terminator is only used to hold physical record terminators with more than one character, e.g. a carriage return and line feed pair. Figure 10-5-1 shows the attributes and methods of RecordHandlerBase class.

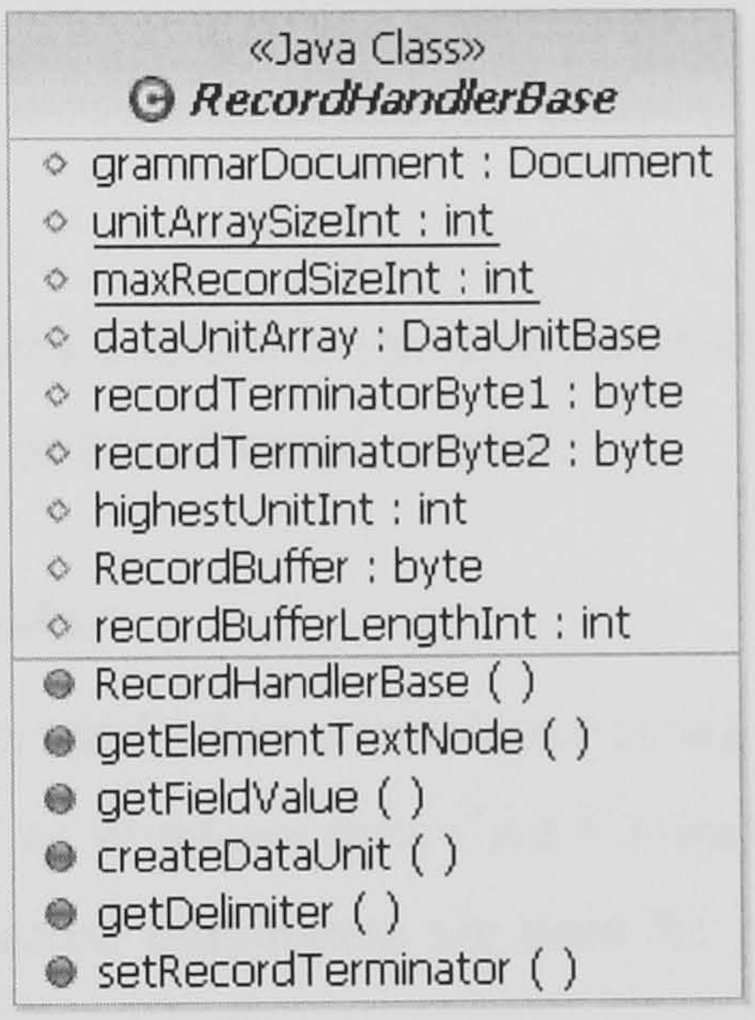

Figure 10-5-1 RecordHandlerBase

10.5.1 Constructor

The constructor method of RecordHandlerBase takes a grammar document as parameter.

It sets up a DataUnit array and initialises other attributes.

10.5.2 createDataUnit

The createDataUnit method takes a field number and a grammar element as parameters, from which a new DataUnit is created, stored in the DataUnit array and returned. The data type of the DataUnit is specified in an attribute of the grammar element.

10.5.3 getElementTextNode

The getElementText method takes an Element as parameter. The comment and empty Nodes of this Element are skipped and the value of Text Node associated with the Element is returned.

10.5.4 getFieldValue

The getFieldValue method takes a field number as parameter. It searches the DataUnit array for the unit with this field number and returns its contents.

10.5.5 getDelimiter

The getDelimiter method takes a delimiter string as parameter. It returns a delimiter as a byte from this passed string or hexadecimal number.

10.5.6 setRecordTerminator

The setRecordTerminator method takes a terminator string as parameter. It sets record terminator as Unix style LF or Windows style CR LF from this passed terminator string or hexadecimal number. Record terminators are used for reading and writing variable length records.

10.6 RecordReaderBase Class

The RecordReaderBase class is the base class of all record reader classes for various legacy formats. It supports reading records from legacy files and transforming them into XML. The attributes and methods are shown in Figure 10-6-1.

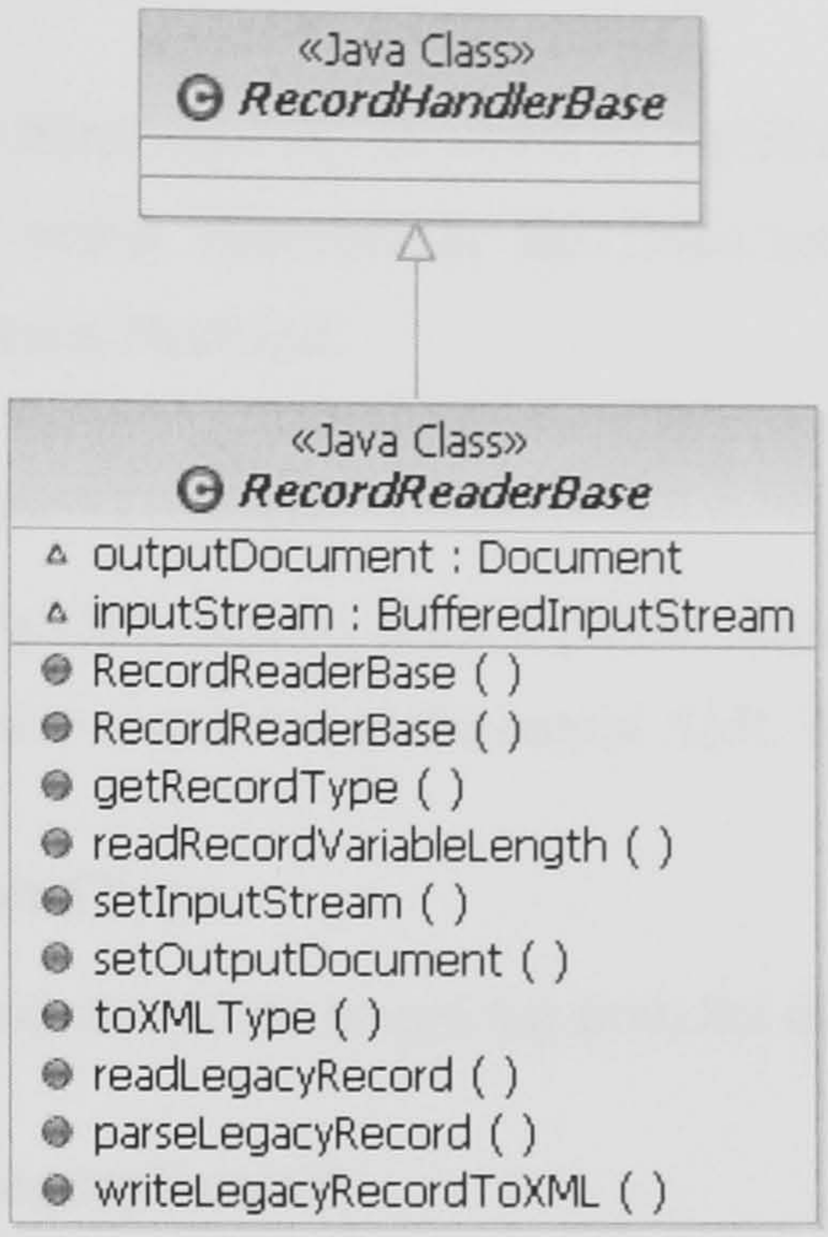

Figure 10-6-1 RecordReaderBase

10.6.1 Constructor

There are two constructors for RecordReaderBase class. One of them takes a grammar document name as parameter and invokes constructor of its base class. The other constructor takes the same parameter and invokes the first constructor with the passed grammar document name. In addition, it takes an input stream as the second parameter and stores it in an attribute.

10.6.2 readRecordVariableLength

The readRecordVariableLenght method reads a record with a variable length from the input file and returns the record length or EOF if no data. The process involves reading one character at a time from the input stream and building the returned input record. The end of a record is determined by the terminator attributes.

10.6.3 setOutputDocument

This is the setter for outputDocument attribute.

10.6.4 toXMLType

The toXMLType method transforms the elements of DataUnit array into XML format by looping through the active elements in the DataUnit array and invoking the convert2XML method of each DataUnit.

10.6.5 writeLegacyRecordToXML

The writeLegacyRecordToXML method takes a parent element and a record grammar element as parameters. It writes a record to the output XML document from a DataUnit.

10.6.6 abstract getRecordType

The getRecordType method returns the record tag from the input record.

10.6.7 abstract parseLegacyRecord

The parseLegacyRecord takes a record grammar element as parameter. It checks the legacy file record and stores the field contents into DataUnit objects of the corresponding subclasses.

10.6.8 abstract readLegacyRecord

The readLegacyRecord method in this base class provides a general interface to subclasses for specific legacy formats for reading physical records.

10.7 RecordWriterBase Class

The RecordWriterBase class is the base class of all record writer classes for legacy formats. It supports writing records from XML documents to legacy files. Figure 10-7-1 shows the attributes and methods of the RecordWriterBase class.

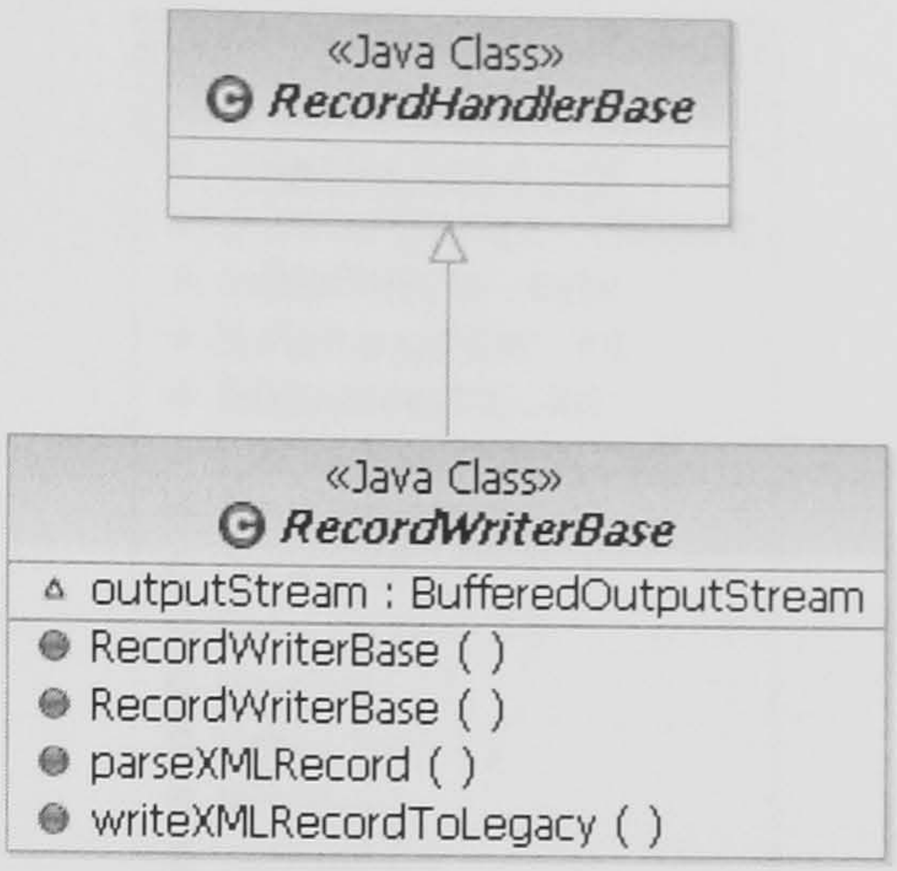

Figure 10-7-1 RecordWriterBase

10.7.1 Constructor

Like RecordReaderBase class, the RecordWriterBase has two constructors. The first constructor takes a single parameter of the grammar document and passes it to the constructor of its base class RecordHandlerBase. The other constructor takes an extra parameter of the output stream and stores it after invoking the base class constructor.

10.7.2 parseXMLRecord

The parseXMLRecord takes a record element and a record grammar element as parameters. It retrieves the child field element nodes from a record element parent and stores its text contents into DataUnit objects of subclasses for specific legacy formats.

10.7.3 abstract writeXMLRecordToLegacy

The writeXMLRecordToLegacy method writes the contents of the DataUnit array to the output legacy record.

10.8 DataUnitBase Class

The DataUnitBase class is the base class for all the classes representing various data types. Subclasses of DataUnitBase support transformations between Schema data types and legacy formats. Figure 10-8-1 shows the attributes and methods of the DataUnitBase class.

Figure 10-8-1 DataUnitBase

10.8.1 Constructor

The constructor takes two the field number and the field grammar element as parameters. It stores these passed arguments in corresponding attributes and initialises other class attributes.

10.8.2 getField

The getField method returns the contents of the Unit Buffer and the value of the Buffer length.

10.8.3 putByte

The putByte method takes a single parameter of byte and appends it to the Unit Buffer.

10.8.4 putField

This method takes two parameters: the contents of a legacy format field of a byte or char array and the length of the field of integer. It stores the passed field contents in the Unit Buffer.

10.8.5 toElement

The toElement method takes a parent element and an output document as parameters. It creates a new element using a grammar element attribute, adds text from the field contents, and attaches the element to the passed parent element.

10.8.6 convert2Legacy

The convert2Legacy method converts the buffer contents from the schema language data type format to the legacy format.

10.8.7 prepareOutput

After format conversion with convert2Legacy, prepareOutput is used for additional formatting tasks required before the data can be written to the output. The operations are specific to data types.

10.8.8 convert2XML

The convert2XML method converts the buffer contents from legacy format to corresponding schema language data type.

CHAPTER 11

Case Study

There are four core techniques proposed in this thesis for Web-based systems evolution: abstraction rules in OCL, architecture description, XML transformation and evolvable Web Application Infrastructure and Framework. In this chapter, case studies have been conducted for these proposed techniques respectively.

11.1 Abstraction and Representation: Tic Tac Toe

To demonstrate the abstraction approach, a case study is conducted on an online game "Noughts and Crosses", shown in Figure 11-1-1. The game allows a person to contact a game server and request a game of noughts and crosses. Once the system has two requests, it pairs the requestors off to play a game and will then wait for further requests, in the meanwhile allowing any existing games to proceed. The server maintains the status of the game and passes moves from one player to the next and also decides who has won the game (or whether it is a draw). It then passes the appropriate messages back to each player.

Figure 11-1-1 Noughts and Crosses

Both the client and server are implemented in Java. Using a system written in Java as the subject of this case study to some extent obviates the need for modules and components identification given the object oriented nature of Java language. There are two reasons for that. Firstly, it is a bit difficult to demonstrate those two phases in one specific example in a general way. Although an overall process and its related activities for those two phases are given in previous chapters, there is often not a standard way of implementing those activities when applying that process in real world. For example, different reverse engineers may have different views concerning which parts of the system should be classified as a component. Secondly, abstraction of individual procedures or methods of a classes and the final representation of the whole architecture is the focus for this time, and those two phases could be implemented automatically without ambiguity.

11.1.1 Abstraction

The fully implemented system is quite complex. Here only one class abstraction is presented, the process of which applies to other classes, too. Although MDA itself is developing rapidly and a number of products claim MDA compliant, reverse engineering a program into MDA models is still quite immature. To date, this research does not intend to present a full fledged tool capable of reversing source code automatically, but to establish a reference model consisting of description languages, abstraction rules and a reengineering process. The following example is well fit for that purpose. The final recovered architecture is represented using ADSR.

```
public class XOServer {
 public static final int DEFAULT_PORT = 3000;
 private Vector sessions;
 private ServerSocket server;
 private boolean switch = false;
 public XOServer() throws IOException {
  sessions = new Vector(10);
  try {
   server = new ServerSocket(DEFAULT_PORT, 20);
   switch = true;
  }
  catch(IOException ioException) {
   ioException.printStackTrace();
   System.exit(1);
  }
}
```

```
public void execute() {
   while (sessions.size()<10) {
    Socket players[] = new Socket[2];
    for (int i=0; i<2; i++) {
     try {
      if (switch) {
       players[i] = server.accept();
      }
     }
     catch(IOException ioException) {
      ioException.printStackTrace();
      System.exit(1);
     }
    }
    sessions.addElement(new GameSession(players[0], players[1]));
   }
  }
}
```

Listing 11-1-1 Source Code of the Server Side Class XOServer

Server side class XOServer waits for two clients to connect and then starts a game between this pair of clients using a GameSession object to create a new thread before returning to wait for more clients. The source code of class XOServer is shown in Listing 11-1-1.

```
public.XOServer = {
 DEFAULT_PORT : public.static.final.int := 3000;
 sessions: private.Vector;
 server: private.ServerSocket;
 switch = false: private.boolean:
 XOServer() {
  sessions := new.Vector(10);
  try {
   server := new.ServerSocket(DEFAULT_PORT, 20);
   switch := true;
  }
  catch(IOException ioException) {
   ioException.printStackTrace();
   System.exit(1);
  }
}
 public void execute() {
  while (sessions.size()<10) {
Socket players[] = new Socket[2];
for (int i=0; i<2; i++) {
 try {
  if (switch) {
   players[i] = server.accept();
  }
 }
```

```
      catch(IOException ioException) {
        ioException.printStackTrace();
        System.exit(1);
      }
    }
    sessions.addElement(new GameSession(players[0], players[1]));
  }
 }
}
```

Listing 11-1-2 PSL Code of the Server Side Class XOServer

Before abstraction, a legacy system is translated into PSL. The PSL code for class XOServer is shown in Listing 11-1-2.

As stated earlier, class XOServer could be regarded as a module that implements specific functionality. We first extract a specification of the class through applying the sequence folding elementary abstraction rules as shown in Listing 11-1-3.

```
public.XOServer  {DEFAULT_PORT,  sessions, server, switch} : {
 XOServer   {}: {
  try {
   server := new.ServerSocket(DEFAULT_PORT, 20);
   switch := true;
  }
  catch(IOException ioException) {
   ioException.printStackTrace();
   System.exit(1);
  }
}
 public.void.execute  {} : {
  {players, i} : ((sessions.size()<10)
                  (
                   ((x<2) (try {
                              (switchOn) (players[i] := server.accept());
                             }
                             catch(IOException ioException) {
                              ioException.printStackTrace();
                              System.exit(1);
                             }
                             x := x + 1
                   ))*
                   sessions.addElement(new.GameSession(players[0], players[1]));
                  ))*
  }
}
```

Listing 11-1-3 XOServer After Appying Elementary Abstraction Rules

State tests and exception handling are often used in programs to assure smooth execution. Although they might be important in system implementation, these details do not involve crucial functionality of a system and in high level specification, these details are unnecessary and should be abstracted away. Therefore in the next step, these details are eliminated by applying another two weakening abstraction rule as shown in Listing 11-1-4.

```
public.XOServer   {DEFAULT_PORT,  sessions, server, switchOn} : {
XOServer    {}: {
 server := new.ServerSocket(DEFAULT_PORT, 20);
 switchOn := true;
}
public.void.execute   {} : {
  {players, i} : ((sessions.size()<10)
                  (
                      ((x<2) (
                                  (switchOn) (players[i] := server.accept());
                                  x := x + 1
                      ))*
                      sessions.addElement(new.GameSession(players[0], players[1]));
                  ))*
 }
}
```

Listing 11-1-4 XOServer After Applying Weakening Abstraction Rules

In the next step, we make the specification more concise and professional by applying domain knowledge, such as patterns, to identify the domain function "socket connection".

11.1.2 Representation

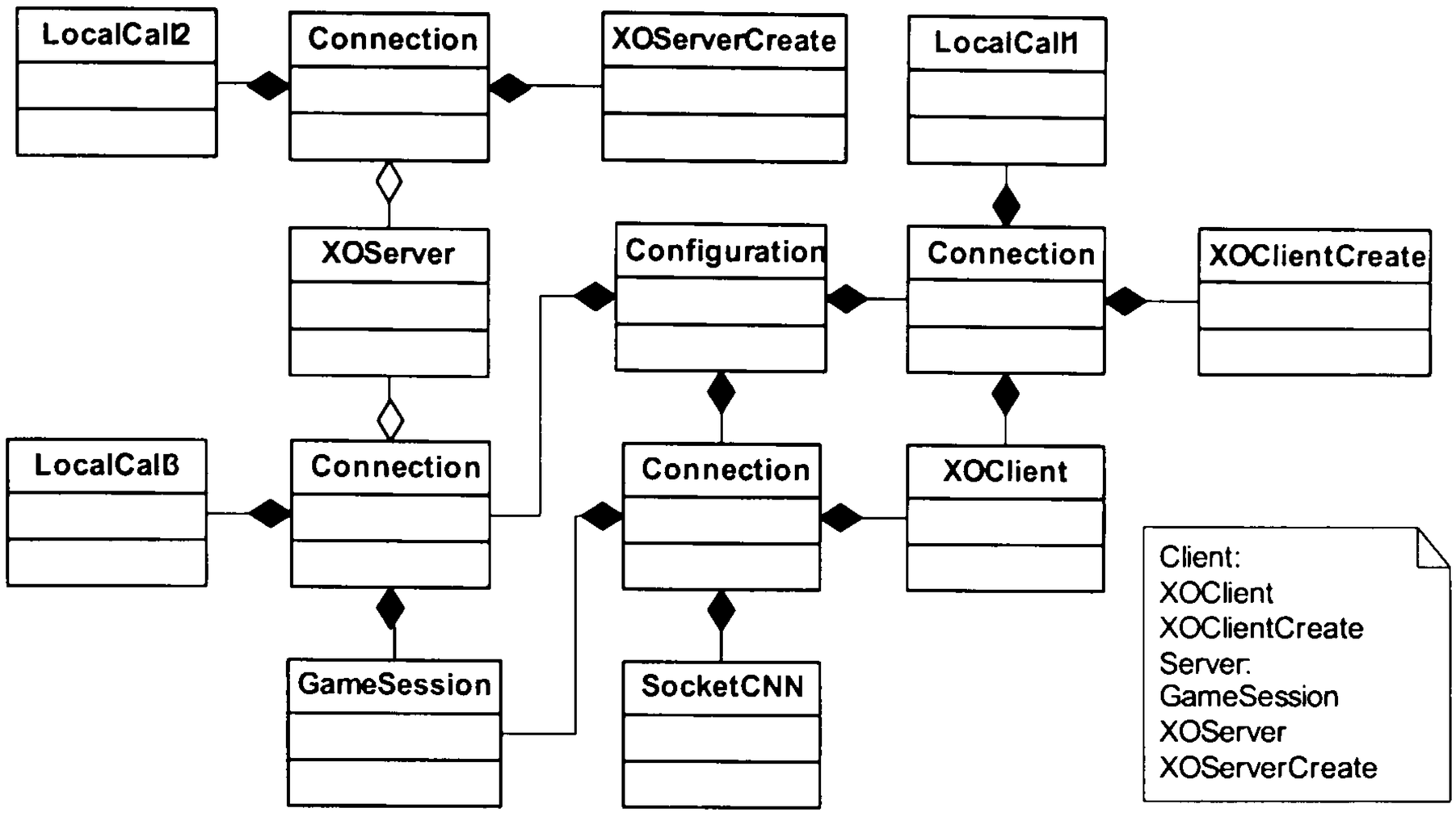

Figure 11-1-2 Architecture Description of Naugts and Crosses

The final architecture of Noughts and Crosses online game consists of three main connectors including two local calls and one socket connection. The configuration of this architecture is composed of three connections which connect their participants respectively. The ADSR representation of this architecture is shown in Figure 11-1-2.

11.2 Legacy Systems Evolution: Web-based XML Reporting Solution

The Comma Separated Value (CSV) file format is widely used for data exchange in heterogeneous applications throughout the industry. An enterprise legacy system often processes many CSV files and needs to publish them as various reports. XML and the Pipes and Filters pattern can integrate such a legacy system with modern reporting tool. This case study will show the transformations between CSV files and XML documents and the information distribution via Web reporting.

11.2.1 Input of CSV to XML: Invoice

- Inputs:

- The first input is a CSV file in uniform Row/Column organisation, where each row has the same column entries and columns are separated by a specified terminator.

- The second input is an XML grammar description document (as discussed in Chapter 9) specifying the CSV file and the grammar of the XML document to be produced.

- Processing: Each input CSV row is transformed into an Element. Each column in the row is transformed into a child Element. Column content is transformed into Schema data types. The elements and Schema data types are all defined in the grammar description document.

- Outputs: One or more XML files are produced, where the root Element name is defined in the grammar description document and the file name is formed by concatenating a three-digit sequence number, the root Element name and the extension .xml.

Column Number	Column Name	Data Type
1	Customer Number	String
2	Invoice Number	String
3	Invoice Date	
4	PO Number	String
5	Due Date	
6	Ship to Name	String
7	Ship to Street 1	String
8	Ship to Street 2	String
9	Ship to City	String
10	Ship to State or Province	String
11	Ship to Postal Code	String
12	Ship to Country	String
13	Item ID	String
14	Item Invoiced Quantity	Int
15	Item Unit Price	double
16	Item Description	String
17	Extended Amount	int

Table 11-2-1 Logical Layout for the Invoice

A simple invoice shown in Table 11-2-1 is used in this case study, where each row of the CSV represents one item to be invoiced. The business is a specialty manufacturer of Computer Cases that sells primarily to Pc hardware chains. Listing 10-2-1 shown below is a sample input invoice that corresponds to the CSV format in Table 11-2-1. Listing 11-2-2 shows the XML Schema for the invoice.

```
CQ123,2005041,10/02/2005,AZ999345,12/2/2005,"PC House - NE Distribution Center","2 Park
Road, NW","",First Avenue,ME,04101,,HCVAN,12,2.59,"CD-R 700m - Grey",30.18
CQ123,2005041,10/02/2005,AZ999345,12/2/2005,"PC House - NE Distribution Center","2 Park
Road, NW","",First Avenue,ME,04101,,HCMIN,24,2.53,"CD-R 700m - Black",58.26
CQ123,2005042,10/02/2005,AW999346,1/12/2006,"PC House - SE Distribution Center","Port
50","3975 Hwy 75",Atoka,OK,74525,,HCVAN,36,2.59,CD-R 700m - Grey,83.14
CQ123,2005042,10/02/2005,AW999346,1/12/2006,"PC House - SE Distribution Center","Port
50","3975 Hwy 75",Atoka,OK,74525,,HCMIN,72,2.53,"CD-R 700m - Black",176.38
WY011,2005043,10/02/2005,2005-0967,1/1/2006,"Trend Software and Hardware","14 Main
Street","",Wichita,KS,67201,,HCVAN,24,2.59,"CD-R 700m - Grey",60.20
CR001,2005044,10/02/2005,5297-0498,12/12/2005,"Simply IT Equipments # 97","37
Waterloo","",Azusa,CA,11102,,HCMIN,120,2.53,"CD-R 700m - Black",299.80
CR001,2005044,10/02/2005,5297-0498,12/12/2005,"Simply IT Equipments # 97","37
Waterloo","",Azusa,CA,11102,,HCVAN,360,2.59,"CD-R 700m - Grey",921.40
CR001,2005044,10/02/2005,5297-0498,12/12/2005,"Simply IT Equipments # 97","37
Waterloo","",Azusa,CA,11102,,HCDUC,240,2.59,"CD-R 700m - Blue",601.50
CR001,2005045,10/02/2005,5245-0498,12/12/2005,"Simply IT Equipments # 45","45 Longroad
46","",Branson,MO,65615,,HCMIN,72,2.53,"CD-R 700m - Black",176.48
CR001,2005045,10/02/2005,5245-0498,12/12/2005,"Simply IT Equipments # 45","45 Longroad
46","",Branson,MO,65615,,HCDUC,96,2.59,"CD-R 700m - Blue",218.40
EQ641,2005046,10/02/2005,987-43671,12/12/2005,"Hawthorn Electronics - DC #1","987
Yorkland Blvd","",Willowdale,ON,M2J 4Y8,CAN,HCMOC,360,2.69,CD-R 700m -
Aluminium,888.40
```

Listing 11-2-1 Input CSV File (Invoices.csv)

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   elementFormDefault="unqualified">
 <xs:element name="Invoice">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="InvoiceLine" maxOccurs="unbounded">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="CustomerNumber" type="xs:token"/>
       <xs:element name="InvoiceNumber" type="xs:token"/>
<xs:element name="InvoiceDate" type="xs:date"/>
<xs:element name="PONumber" type="xs:token"/>
<xs:element name="DueDate" type="xs:date"/>
<xs:element name="ShipToName" type="xs:token"/>
<xs:element name="ShipToStreet1" type="xs:token"/>
<xs:element name="ShipToStreet2" type="xs:token"
   minOccurs="0"/>
```

```
<xs:element name="ShipToCity" type="xs:token"/>
<xs:element name="ShipToStateOrProvince"
  type="xs:token"/>
<xs:element name="ShipToPostalCode"
  type="xs:token"/>
<xs:element name="ShipToCountry" type="xs:token"
  minOccurs="0"/>
<xs:element name="ItemID" type="xs:token"/>
<xs:element name="ItemQuantity"
  type="xs:positiveInteger"/>
<xs:element name="UnitPrice" type="xs:decimal"/>
<xs:element name="ItemDescription"
  type="xs:token"/>
        <xs:element name="ExtendedPrice"
          type="xs:decimal"/>
       </xs:sequence>
      </xs:complexType>
     </xs:element>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
</xs:schema>
```

Listing 11-2-2 Invoice Schema (CSVInvoice.xsd)

11.2.2 Input of XML to CSV: Purchase Order

- Inputs:
  - The first input is one or more XML files are transformed into CSV files.
  - The second input is an XML grammar description document (as discussed in Chapter 9) specifying the XML documents and the CSV file to be produced.
- Processing: According to the grammar description document
  - Each input Element corresponding to a row is transformed into a record in the CSV output file.
  - Each of its child Elements corresponding to columns are transformed into the appropriate column numbers.
  - Column delimiter characters are added and data types are converted according to the column descriptions.

- Output: A CSV file in row and column organisation.

Column Number	Column Name	Data Type
1	Customer Number	String
2	PO Number	String
3	PO Date	Date
4	Requested Delivery Date	Date
5	Ship to Name	String
6	Ship to Street 1	String
7	Ship to Street 2	String
8	Ship to City	String
9	Ship to State or Province	String
10	Ship to Postal Code	String
11	Ship to Country	String
12	Item ID	String
13	Item Quantity	int
14	Item Unit Price	double
15	Item Description	String

Table 11-2-2 Logical Layout for the Purchase Order

Defined in Table 11-2-2, a purchase order is used as the sample input XML document. Three purchase orders that follow this logical organisation are shown in Listing 11-2-3 to Listing 11-2-5. Listing 11-2-6 shows the XML Schema for the purchase order.

```
<?xml version="1.0" encoding="UTF-8"?>
<PurchaseOrder>
 <POLine>
  <CustomerNumber>BQ003</CustomerNumber>
  <PONumber>AZ999345</PONumber>
  <PODate>2005-05-06</PODate>
  <RequestedDeliveryDate>2005-06-06</RequestedDeliveryDate>
  <ShipToName>
   PC House - NE Distribution Center
  </ShipToName>
  <ShipToStreet1>12 Industrial Parkway, NW</ShipToStreet1>
  <ShipToCity>Portland</ShipToCity>
  <ShipToState>ME</ShipToState>
  <ShipToPostalCode>04101</ShipToPostalCode>
  <ItemID>HCVAN</ItemID>
  <OrderedQty>12</OrderedQty>
  <UnitPrice>2.59</UnitPrice>
  <ItemDescription>
   CD-R 700m - Grey
  </ItemDescription>
</POLine>
<POLine>
  <CustomerNumber>BQ003</CustomerNumber>
  <PONumber>AZ999345</PONumber>
```

```
<PODate>2005-05-06</PODate>
<RequestedDeliveryDate>2005-06-06</RequestedDeliveryDate>
<ShipToName>
 PC House - NE Distribution Center
</ShipToName>
<ShipToStreet1>12 Industrial Parkway, NW</ShipToStreet1>
<ShipToCity>Portland</ShipToCity>
<ShipToState>ME</ShipToState>
<ShipToPostalCode>04101</ShipToPostalCode>
<ItemID>HCMIN</ItemID>
<OrderedQty>24</OrderedQty>
<UnitPrice>2.53</UnitPrice>
<ItemDescription>
  CD-R 700m - Black
 </ItemDescription>
</POLine>
</PurchaseOrder>
```

Listing 11-2-3 Purchase Order 1

```
<?xml version="1.0" encoding="UTF-8"?>
<PurchaseOrder>
 <POLine>
  <CustomerNumber>BQ003</CustomerNumber>
  <PONumber>AW999346</PONumber>
  <PODate>2005-05-06</PODate>
  <RequestedDeliveryDate>2005-06-06</RequestedDeliveryDate>
  <ShipToName>
   PC House - SE Distribution Center
  </ShipToName>
  <ShipToStreet1>Dock 37</ShipToStreet1>
  <ShipToStreet2>3975 Hwy 75</ShipToStreet2>
  <ShipToCity>Atoka</ShipToCity>
  <ShipToState>OK</ShipToState>
  <ShipToPostalCode>74525</ShipToPostalCode>
  <ItemID>HCVAN</ItemID>
  <OrderedQty>36</OrderedQty>
  <UnitPrice>2.59</UnitPrice>
  <ItemDescription>
   CD-R 700m - Grey
  </ItemDescription>
 </POLine>
 <POLine>
  <CustomerNumber>BQ003</CustomerNumber>
  <PONumber>AW999346</PONumber>
  <PODate>2005-05-06</PODate>
  <RequestedDeliveryDate>2005-06-06</RequestedDeliveryDate>
  <ShipToName>
   PC House - SE Distribution Center
  </ShipToName>
  <ShipToStreet1>Dock 37</ShipToStreet1>
  <ShipToStreet2>3975 Hwy 75</ShipToStreet2>
  <ShipToCity>Atoka</ShipToCity>
  <ShipToState>OK</ShipToState>
```

```
   <ShipToPostalCode>74525</ShipToPostalCode>
   <ItemID>HCMIN</ItemID>
   <OrderedQty>72</OrderedQty>
   <UnitPrice>2.53</UnitPrice>
   <ItemDescription>
    CD-R 700m - Black
   </ItemDescription>
  </POLine>
</PurchaseOrder>
```

Listing 11-2-4 Purchase Order 2

```
<?xml version="1.0" encoding="UTF-8"?>
<PurchaseOrder>
 <POLine>
  <CustomerNumber>AY001</CustomerNumber>
  <PONumber>2002-0967</PONumber>
  <PODate>2005-05-06</PODate>
  <RequestedDeliveryDate>2005-06-04</RequestedDeliveryDate>
  <ShipToName>Trend Software and Hardware</ShipToName>
  <ShipToStreet1>14 Main Street</ShipToStreet1>
  <ShipToCity>Wichita</ShipToCity>
  <ShipToState>KS</ShipToState>
  <ShipToPostalCode>67201</ShipToPostalCode>
  <ItemID>HCVAN</ItemID>
  <OrderedQty>24</OrderedQty>
  <UnitPrice>2.59</UnitPrice>
  <ItemDescription>
   CD-R 700m - Grey
  </ItemDescription>
 </POLine>
</PurchaseOrder>
```

Listing 11-2-5 Purchase Order 3

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="unqualified">
 <xs:element name="PurchaseOrder">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="POLine" maxOccurs="unbounded">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="CustomerNumber" type="xs:token"/>
       <xs:element name="PONumber" type="xs:token"/>
<xs:element name="PODate" type="xs:date"/>
<xs:element name="RequestedDeliveryDate"
  type="xs:date"/>
<xs:element name="ShipToName" type="xs:token"/>
<xs:element name="ShipToStreet1" type="xs:token"/>
<xs:element name="ShipToStreet2" type="xs:token"
  minOccurs="0"/>
```

```
<xs:element name="ShipToCity" type="xs:token"/>
<xs:element name="ShipToStateOrProvince"
   type="xs:token"/>
<xs:element name="ShipToPostalCode"
   type="xs:token"/>
<xs:element name="ShipToCountry" type="xs:token"
   minOccurs="0"/>
<xs:element name="ItemID">
 <xs:simpleType>
  <xs:restriction base="xs:token">
   <xs:enumeration value="HCDUC"/>
   <xs:enumeration value="HCMIN"/>
   <xs:enumeration value="HCMOC"/>
   <xs:enumeration value="HCVAN"/>
  </xs:restriction>
 </xs:simpleType>
</xs:element>
<xs:element name="OrderedQty"
   type="xs:positiveInteger"/>
<xs:element name="UnitPrice" type="xs:decimal"
   minOccurs="0"/>
<xs:element name="ItemDescription" type="xs:token"
   minOccurs="0"/>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

Listing 11-2-6 Purchase Order Schema (CSVPurchaseOrder.xsd)

11.2.3 Grammar Description and Schema

The grammar of legacy file formats can be divided into two sets: the grammar of records and groups of records within the file, and the grammar of the fields within a record.

11.2.3.1 CSV File Grammar

Each row of CSV files used in this case study has the same format. The CSV file grammar can be defined with the following BNF production.

```
CSV ::= row+
```

Listing 11-2-7 CSV File Grammar

11.2.3.2 CSV Row Grammar

The row nonterminal symbol from the file grammar is defined in the CSV row grammar shown in Listing 11-2-8.

```
row ::= column (column_delimiter column?)* | (column_delimiter column?)+
column ::= column_characters_A+ | text_delimiter column_characters_B+ text_delimiter
column_characters_A ::= All allowed characters except column_delimiter
column_characters_B ::= All allowed characters except text_delimiter
```

Listing 11-2-8 CSV Row Grammar

11.2.3.3 Grammar Description Document Schemas

There are three schemas presented in this subsection, which are given in Appendix B.

- CSVSourceGrammarDescription.xsd specifies transformations where the source format is a CSV file.

- CSVTargetGrammarDescription.xsd specifies transformations where the target format is a CSV file.

- CSVCommonGrammarDescription.xsd specifies type library used by the above two schemas.

11.2.4 Output of CSV to XML

Listing 11-2-9 shows the XML document converted from the CSV file shown in Listing 11-2-1.

```
<?xml version="1.0" encoding="UTF-8"?>
<Invoice xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <InvoiceLine>
    <CustomerNumber>CQ123</CustomerNumber>
    <InvoiceNumber>2005041</InvoiceNumber>
    <InvoiceDate>2005-10-02</InvoiceDate>
    <PONumber>AZ999345</PONumber>
    <DueDate>2005-12-02</DueDate>
    <ShipToName>PC House - NE Distribution Center</ShipToName>
    <ShipToStreet1>2 Park Road, NW</ShipToStreet1>
    <ShipToCity>First Avenue</ShipToCity>
    <ShipToStateOrProvince>ME</ShipToStateOrProvince>
    <ShipToPostalCode>04101</ShipToPostalCode>
    <ItemID>HCVAN</ItemID>
    <ItemQuantity>12</ItemQuantity>
```

```
    <UnitPrice>2.59</UnitPrice>
    <ItemDescription>CD-R 700m - Grey</ItemDescription>
    <ExtendedPrice>30.18</ExtendedPrice>
</InvoiceLine>
<InvoiceLine>
    <CustomerNumber>CQ123</CustomerNumber>
    <InvoiceNumber>2005041</InvoiceNumber>
    <InvoiceDate>2005-10-02</InvoiceDate>
    <PONumber>AZ999345</PONumber>
    <DueDate>2005-12-02</DueDate>
    <ShipToName>PC House - NE Distribution Center</ShipToName>
    <ShipToStreet1>2 Park Road, NW</ShipToStreet1>
    <ShipToCity>First Avenue</ShipToCity>
    <ShipToStateOrProvince>ME</ShipToStateOrProvince>
    <ShipToPostalCode>04101</ShipToPostalCode>
    <ItemID>HCMIN</ItemID>
    <ItemQuantity>24</ItemQuantity>
    <UnitPrice>2.53</UnitPrice>
    <ItemDescription>CD-R 700m - Black</ItemDescription>
    <ExtendedPrice>58.26</ExtendedPrice>
  </InvoiceLine>
</Invoice>

<?xml version="1.0" encoding="UTF-8"?>
<Invoice xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <InvoiceLine>
    <CustomerNumber>CQ123</CustomerNumber>
    <InvoiceNumber>2005042</InvoiceNumber>
    <InvoiceDate>2005-10-02</InvoiceDate>
    <PONumber>AW999346</PONumber>
    <DueDate>2006-01-12</DueDate>
    <ShipToName>PC House - SE Distribution Center</ShipToName>
    <ShipToStreet1>Port 50</ShipToStreet1>
    <ShipToStreet2>3975 Hwy 75</ShipToStreet2>
    <ShipToCity>Atoka</ShipToCity>
    <ShipToStateOrProvince>OK</ShipToStateOrProvince>
    <ShipToPostalCode>74525</ShipToPostalCode>
    <ItemID>HCVAN</ItemID>
    <ItemQuantity>36</ItemQuantity>
    <UnitPrice>2.59</UnitPrice>
    <ItemDescription>CD-R 700m - Grey</ItemDescription>
    <ExtendedPrice>83.14</ExtendedPrice>
</InvoiceLine>
<InvoiceLine>
    <CustomerNumber>CQ123</CustomerNumber>
    <InvoiceNumber>2005042</InvoiceNumber>
    <InvoiceDate>2005-10-02</InvoiceDate>
<PONumber>AW999346</PONumber>
<DueDate>2006-01-12</DueDate>
<ShipToName>PC House - SE Distribution Center</ShipToName>
<ShipToStreet1>Port 50</ShipToStreet1>
<ShipToStreet2>3975 Hwy 75</ShipToStreet2>
<ShipToCity>Atoka</ShipToCity>
<ShipToStateOrProvince>OK</ShipToStateOrProvince>
```

```
      <ShipToPostalCode>74525</ShipToPostalCode>
      <ItemID>HCMIN</ItemID>
      <ItemQuantity>72</ItemQuantity>
      <UnitPrice>2.53</UnitPrice>
      <ItemDescription>CD-R 700m - Black</ItemDescription>
      <ExtendedPrice>176.38</ExtendedPrice>
   </InvoiceLine>
</Invoice>

<?xml version="1.0" encoding="UTF-8"?>
<Invoice xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <InvoiceLine>
      <CustomerNumber>WY011</CustomerNumber>
      <InvoiceNumber>2005043</InvoiceNumber>
      <InvoiceDate>2005-10-02</InvoiceDate>
      <PONumber>2005-0967</PONumber>
      <DueDate>2006-01-01</DueDate>
      <ShipToName>Trend Software and Hardware</ShipToName>
      <ShipToStreet1>14 Main Street</ShipToStreet1>
      <ShipToCity>Wichita</ShipToCity>
      <ShipToStateOrProvince>KS</ShipToStateOrProvince>
      <ShipToPostalCode>67201</ShipToPostalCode>
      <ItemID>HCVAN</ItemID>
      <ItemQuantity>24</ItemQuantity>
      <UnitPrice>2.59</UnitPrice>
      <ItemDescription>CD-R 700m - Grey</ItemDescription>
      <ExtendedPrice>60.20</ExtendedPrice>
   </InvoiceLine>
</Invoice>

<?xml version="1.0" encoding="UTF-8"?>
<Invoice xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <InvoiceLine>
      <CustomerNumber>CR001</CustomerNumber>
      <InvoiceNumber>2005044</InvoiceNumber>
      <InvoiceDate>2005-10-02</InvoiceDate>
      <PONumber>5297-0498</PONumber>
      <DueDate>2005-12-12</DueDate>
      <ShipToName>Simply IT Equipments # 97</ShipToName>
      <ShipToStreet1>37 Waterloo</ShipToStreet1>
      <ShipToCity>Azusa</ShipToCity>
      <ShipToStateOrProvince>CA</ShipToStateOrProvince>
      <ShipToPostalCode>11102</ShipToPostalCode>
      <ItemID>HCMIN</ItemID>
      <ItemQuantity>120</ItemQuantity>
      <UnitPrice>2.53</UnitPrice>
      <ItemDescription>CD-R 700m - Black</ItemDescription>
      <ExtendedPrice>299.80</ExtendedPrice>
   </InvoiceLine>
   <InvoiceLine>
      <CustomerNumber>CR001</CustomerNumber>
      <InvoiceNumber>2005044</InvoiceNumber>
      <InvoiceDate>2005-10-02</InvoiceDate>
      <PONumber>5297-0498</PONumber>
```

```
    <DueDate>2005-12-12</DueDate>
    <ShipToName>Simply IT Equipments # 97</ShipToName>
    <ShipToStreet1>37 Waterloo</ShipToStreet1>
    <ShipToCity>Azusa</ShipToCity>
    <ShipToStateOrProvince>CA</ShipToStateOrProvince>
    <ShipToPostalCode>11102</ShipToPostalCode>
    <ItemID>HCVAN</ItemID>
    <ItemQuantity>360</ItemQuantity>
    <UnitPrice>2.59</UnitPrice>
    <ItemDescription>CD-R 700m - Grey</ItemDescription>
    <ExtendedPrice>921.40</ExtendedPrice>
</InvoiceLine>
<InvoiceLine>
      <CustomerNumber>CR001</CustomerNumber>
      <InvoiceNumber>2005044</InvoiceNumber>
      <InvoiceDate>2005-10-02</InvoiceDate>
      <PONumber>5297-0498</PONumber>
      <DueDate>2005-12-12</DueDate>
      <ShipToName>Simply IT Equipments # 97</ShipToName>
      <ShipToStreet1>37 Waterloo</ShipToStreet1>
      <ShipToCity>Azusa</ShipToCity>
      <ShipToStateOrProvince>CA</ShipToStateOrProvince>
      <ShipToPostalCode>11102</ShipToPostalCode>
      <ItemID>HCDUC</ItemID>
      <ItemQuantity>240</ItemQuantity>
      <UnitPrice>2.59</UnitPrice>
      <ItemDescription>CD-R 700m - Blue</ItemDescription>
      <ExtendedPrice>601.50</ExtendedPrice>
  </InvoiceLine>
</Invoice>

<?xml version="1.0" encoding="UTF-8"?>
<Invoice xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <InvoiceLine>
    <CustomerNumber>CR001</CustomerNumber>
    <InvoiceNumber>2005045</InvoiceNumber>
    <InvoiceDate>2005-10-02</InvoiceDate>
    <PONumber>5245-0498</PONumber>
    <DueDate>2005-12-12</DueDate>
    <ShipToName>Simply IT Equipments # 45</ShipToName>
    <ShipToStreet1>45 Longroad 46</ShipToStreet1>
    <ShipToCity>Branson</ShipToCity>
    <ShipToStateOrProvince>MO</ShipToStateOrProvince>
    <ShipToPostalCode>65615</ShipToPostalCode>
    <ItemID>HCMIN</ItemID>
    <ItemQuantity>72</ItemQuantity>
    <UnitPrice>2.53</UnitPrice>
    <ItemDescription>CD-R 700m - Black</ItemDescription>
    <ExtendedPrice>176.48</ExtendedPrice>
</InvoiceLine>
<InvoiceLine>
    <CustomerNumber>CR001</CustomerNumber>
    <InvoiceNumber>2005045</InvoiceNumber>
    <InvoiceDate>2005-10-02</InvoiceDate>
```

```
    <PONumber>5245-0498</PONumber>
    <DueDate>2005-12-12</DueDate>
    <ShipToName>Simply IT Equipments # 45</ShipToName>
    <ShipToStreet1>45 Longroad 46</ShipToStreet1>
    <ShipToCity>Branson</ShipToCity>
    <ShipToStateOrProvince>MO</ShipToStateOrProvince>
    <ShipToPostalCode>65615</ShipToPostalCode>
    <ItemID>HCDUC</ItemID>
    <ItemQuantity>96</ItemQuantity>
    <UnitPrice>2.59</UnitPrice>
    <ItemDescription>CD-R 700m - Blue</ItemDescription>
    <ExtendedPrice>218.40</ExtendedPrice>
  </InvoiceLine>
</Invoice>

<?xml version="1.0" encoding="UTF-8"?>
<Invoice xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <InvoiceLine>
    <CustomerNumber>EQ641</CustomerNumber>
    <InvoiceNumber>2005046</InvoiceNumber>
    <InvoiceDate>2005-10-02</InvoiceDate>
    <PONumber>987-43671</PONumber>
    <DueDate>2005-12-12</DueDate>
    <ShipToName>Hawthorn Electronics - DC #1</ShipToName>
    <ShipToStreet1>987 Yorkland Blvd</ShipToStreet1>
    <ShipToCity>Willowdale</ShipToCity>
    <ShipToStateOrProvince>ON</ShipToStateOrProvince>
    <ShipToPostalCode>M2J 4Y8</ShipToPostalCode>
    <ShipToCountry>CAN</ShipToCountry>
    <ItemID>HCMOC</ItemID>
    <ItemQuantity>360</ItemQuantity>
    <UnitPrice>2.69</UnitPrice>
    <ItemDescription>CD-R 700m - Aluminium</ItemDescription>
    <ExtendedPrice>888.40</ExtendedPrice>
  </InvoiceLine>
</Invoice>
```

Listing 11-2-9 Output Documents

11.2.5 Output of XML to CSV

Listing 11-2-10 is the CSV file converted from XML document shown in Listing 11-2-3, 11-2-4 and 11-2-5.

```
BQ003,AZ999345,06/05/2005,06/06/2005, "PC House - NE Distribution Center", "12 Industrial
Parkway, NW",,"Portland",ME,04101,, HCVAN,12,2.59,"CD-R 700m - Grey"
BQ003,AZ999345,06/05/2005,06/06/2005, "PC House - NE Distribution Center", "12 Industrial
Parkway, NW",,"Portland",ME,04101,, HCMIN,24,2.53,"CD-R 700m - Black"
BQ003,AW999346,06/05/2005,06/06/2005, "PC House - SE Distribution Center", "Dock
37","3975 Hwy 75","Atoka",OK,74525,, HCVAN,36,2.59,"CD-R 700m - Grey"
BQ003,AW999346,06/05/2005,06/06/2005, "PC House - SE Distribution Center", "Dock
37","3975 Hwy 75","Atoka",OK,74525,, HCMIN,72,2.53,"CD-R 700m - Black"
```

AY001,2002-0967,06/05/2005,04/06/2005, "Trend Software and Hardware", "14 Main Street",,"Wichita",KS,67201,, HCVAN,24,2.59,"CD-R 700m - Grey"

Listing 11-2-10 Purchase Order 1

11.2.6 Web-based XML Distribution

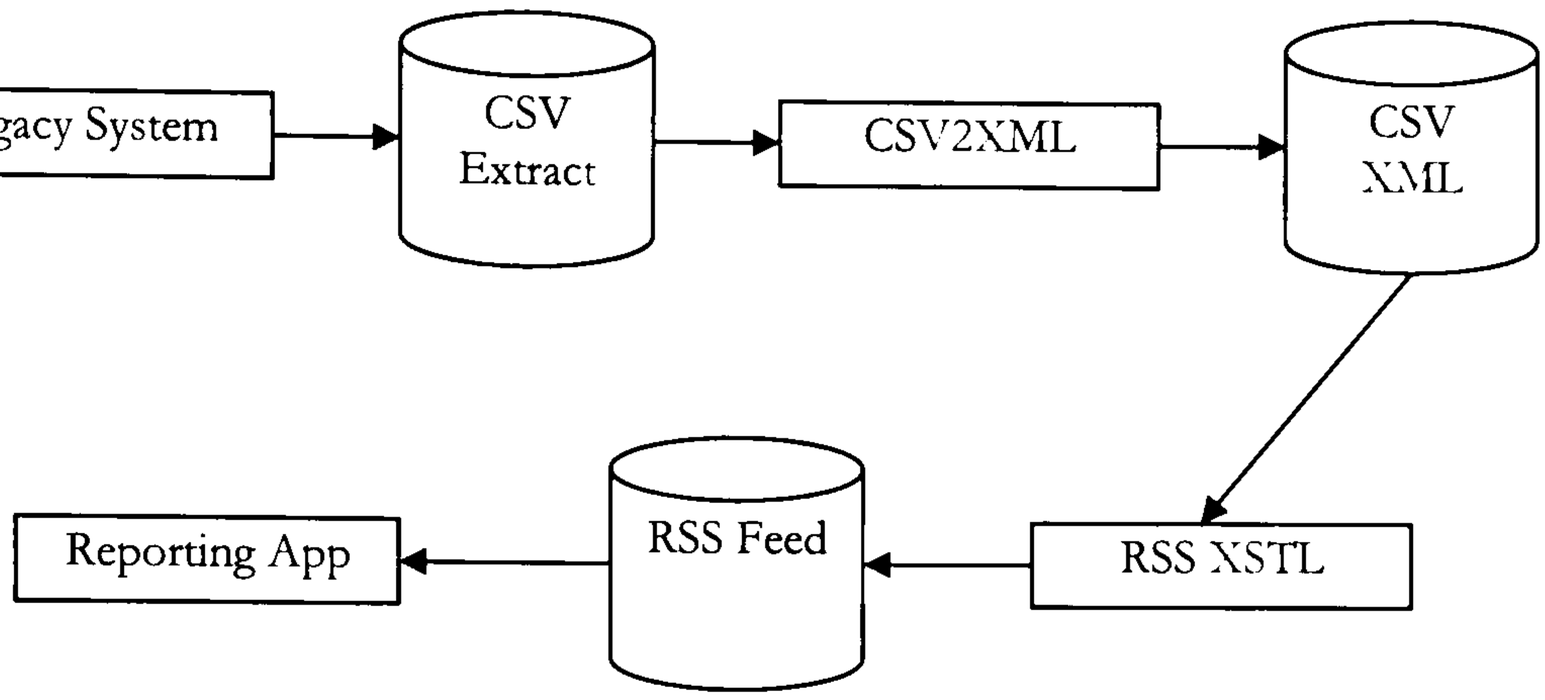

Figure 11-2-11 CSV to RSS Feed

The produced XML documents can be integrated as RSS feeds to be published via Web as shown in Figure 11-2-11.

11.3 Proposed Web Application Infrastructure and Framework: PetStore Online

The application used to demonstrate the proposed Web Application Infrastructure and framework is an online pet store. The pet store currently sells pets of three categories, each of which has a default delivery plan, used for most customers, but occasionally a particular customer may vary this slightly. The delivery plan for foreign customers might need extra work in accordance with export regulations.

To appeal new visitors and retain existing customers, it must be possible to change the presentation of the site to another style without changing the basic functionality. In addition, some of site's visitors are non-English speakers, so there might be requirement for internationalisation. Finally the pet store currently has no online ticketing system. Little of the existing order processing system, except parts of the database schema, is likely to be reusable for the web interface.

11.3.1 Business Logic Implementation

This section will discuss how the PetStore application uses standard J2EE infrastructure and how it uses the infrastructure proposed in this research. The key business logic in this application is: how to obtain reference data about categories, breeds and descriptions; and how to obtain information about stock availability and implement the ordering process.

11.3.1.1 Business Interfaces

Business interfaces should not be web-specific. Two separate interfaces are required to handle reference data and ordering process respectively.

The org.petstore.pet.referencedata.Inventory interface shown in Listing 11-3-1 handles reference data requirements.

```
public interface Inventory {
  List getCategories ();
  Breed getBreed(int id) throws ReferenceDataException;
  Description getDescription(int id) throws ReferenceDataException;
}
```

Listing 11-3-1 Inventory Interface

The Category, Breed and Description objects are value objects independent from the data source. The org.petstore.pet.orderprocessor.OrderProcessor interface shown in Listing 11-3-2 exposes information about stock level and the ordering process.

```
public interface OrderProcessor {

  int getStock (int petId) throws NoSuchPetException;

  Order placeOrder (OrderRequest request)
    throws NotEnoughStockException, NoSuchPetException;

  Invoice confirmOrder (PurchaseRequest purchaseRequest)
    CreditCardAuthorizationException,
    InvalidPetRequestException,
    OrderProcessorInternalException;
}
```

Listing 11-3-2 OrderProcessor Interface

The method exposes stock level. The placeOrder() method creates an order, while the confirmOrder() method turns an existing order into a purchase. The OrderRequest object, used as a parameter to the placeOrder() method, is a command, as is the PurchaseRequest parameter to the confirmOrder() method. The use of command objects are used in place of multiple individual parameters to provide flexibility in adding parameter data without breaking method signatures.

11.3.1.2 Implementation Strategy

The Inventory interface involves non-transactional data access. Thus DAOs (Data Access Object) in the web container should suffice instead of implementing it as EJB. The OrderProcessor interface involves transactional data manipulation, which can be implemented as a stateless session bean with a local interface, where CMT (Container Managed Transaction) can be used to simplify application code. Such decisions will not affect the calling code of these objects. Thus adopting a different data-access strategy will have little impact on the existing system.

The org.petstore.pet.referencedata.jdbc.JdbcInventory class implements the org.evolvable.beans.factory.InitializingBean interface, enabling it to load data in its initialization of the afterPropertiesSet() method.

11.3.1.3 Overall Architecture of PetStore

This section demonstrates how to wire up the interfaces and implementations defined earlier via the proposed infrastructure. Listing 11-3-3 shows code snippets from the pet-servlet.xml application registry XML definition file for two bean definitions.

```
<bean name="realInventory"
 singleton="false"
 class="org.petstore.pet.referencedata.jdbc.JdbcInventory">
</bean>

<bean name="inventory"
 class="org.petstore.pet.referencedata.support.CachingInventory">
</bean>
```

Listing 11-3-3 Bean Definitions from Application Registry

The OrderProcessor bean definition uses the dynamic proxy bean definition to hide the complexity of EJB access shown in Listing 11-3-4.

```
<bean name="orderProcessor"
    definitionClass="
     org.evolvable.ejb.access.LocalStatelessSessionProxyBeanDefinition">

 <customDefinition property="businessInterface">
   org.petstore.pet.orderprocessor.OrderProcessor
 </customDefinition>
 <customDefinition property="jndiName">
   ejb/OrderProcessor
 </customDefinition>

</bean>
```

Listing 11-3-4 OrderProcessor Bean Definition

Application objects do not need to perform JNDI lookups or use the EJB API directly, as they can use the object created by this bean definition as an implementation of the org.petstore.pet.orderprocessor.OrderProcessor business interface. Application components using these interfaces need simply expose bean properties that can be set by the proposed infrastructure automatically. They have no dependency on infrastructure code and do not need to look up objects managed by the infrastructure. For example, the application's main web-tier controller exposes inventory and orderProcessor properties, which the infrastructure sets automatically based on bean definition shown in Listing 11-3-5.

```
<bean name="petController"
 class="org.petstore.pet.web.PetController">

 // Other properties omitted

 <property name="inventory" beanRef="true">inventory</property>
 <property name="orderProcessor" beanRef="true">orderProcessor</property>

</bean>
```

Listing 11-3-5 Bean Definition for petController

The controller does not need any configuration lookup, but merely has to implement the JavaBean setters shown in Listing 11-3-6.

```
public void setOrderProcessor (OrderProcessor orderProcessor) {
  this. orderProcessor = orderProcessor;
}

public void setInventory (Inventory inventory) {
  this. inventory = inventory;
}
```

Listing 11-3-6 JavaBean Setters for Controller Configuration

11.3.2 Web Tier Implementation

The Web tier in this case study is implemented using proposed WAF to achieve the similar functionality of Pet Store, but a cleaner and thinner architecture. The configuration of the Web tier is as follows:

- The Web interface contains a single controller PetController that extends the Method Mapping Controller to handle all application URLs.

- View is implemented as JSP pages, which render non-Web-specific model beans returned by the controller.

- Defining the ControllerServlet (with name pet) and RegistryLoaderServlet in the web.xml file.

- Creating the pet-servlet.xml XML application registry definition file specifying the beans required by the controller servlet, including:

    - Definitions of business object beans.

    - Definition of the PetController web controller bean, setting its bean properties.

    - A ControllerMapping object that maps all application URLs onto the controller bean.

Listing 11-3-7 shows the complete definition of the PetController bean in the application's /WEB-INF/pet-servlet.xml file:

```
<bean name="petController" class="org.petstore.pet.web.PetController ">
```

```
<property name="methodMapper" beanRef="true">
    petControllerMethodMapper
</property>
<property name="calendar" beanRef="true">
    calendar
</property>
<property name="orderProcessor" beanRef="true">
    branch
</property>
<property name="inventoryCheck" beanRef="true">
    inventoryCheck
</property>
<property name="validator" beanRef="true">
    validator
  </property>
  <property name="deliveryCharge">5.50</property>
</bean>
```

Listing 11-3-7 PetController Definition

Mappings from request URL to controller bean name are defined via a standard framework class as follows:

```
<bean name="a.urlMap" class="UrlControllerMapping">
    <property name="mappings">
        /homepage.html=petController
        /breed.html= petController
        /order.html= petController
        /checkout.html= petController
        /confirm.html= petController
    </property>
</bean>
```

Listing 11-3-8 ControllerMapping Definition

The controller handles all requests, which are mapped onto individual methods of PetController class via its methodMapper property inherited from MethodMappingController, and defines the mapping of request URLs to individual methods in the PetController class.

```
<bean name="petControllerMethodMapper" class="PropertiesMethodMapper">
  <property name="mappings">
        /homepage.html=showCategoriesPage
        /breed.html=showBreed
        /order.html=showOrderForm
        /checkout.html=showCheckoutForm
        /confirm.html=confirmCheckoutFormSubmission
  </property>
</bean>
```

Listing 11-3-9 MethodMapper Definition

11.3.2.1 Handling a Order Placement Request

As error handling on invalid input can be implemented via the automatic data binding capability of the MethodMappingController superclass. Listing 11-3-10 shows the request handling method for order request.

```
public ModelAndView processOrderFormSubmission(
HttpServletRequest request,
  HttpServletResponse response,
  OrderRequest orderRequest)
  throws ServletException,
       InvalidOrderRequestException,
       NoSuchCategoryException {
...
}
```

Listing 11-3-10 Request Handling

Invoking this method will populate the OrderRequest object with request parameters containing information for the order: pet type ID, category ID and the number of pets requested. All the following instance variables in the PetController class are set via the bean properties set in the XML bean definition element:

```
orderRequest.setDeliveryCharge(this.deliveryCharge);
orderRequest.setReserve(true);
```

Listing 11-3-11 Bean Properties Initialised from XML Configuration File

With a fully configured OrderRequest object, user session can be checked to see if there is already an order matching the request.

```
Order order = null;
HttpSession session = request.getSession(false);
if (session != null) {
  order = (Order) session.getAttribute(ORDER_KEY);
  if (order != null) {
    if (order.verifyRequest(orderRequest)) {
     return new ModelAndView("showOrder", ORDER_KEY, order);
     } else {
      order = null;
      session.removeAttribute(ORDER_KEY);
     }
    }
}
```

Listing 11-3-12 Showing an Existing Order

Invoking the OrderProcessor business object will create an Order. The orderProcessor instance variable of PetController is set via a bean property during application startup, and its placeOrder() method will either return an Order or throw a NotEnoughInventoryException, which will be caught and prompt the display of a different view. A successfully created order will be stored in the user's session.

```
try {
  order = orderProcessor.placeOrder(orderRequest);
  session = request.getSession(true);
  session.setAttribute(ORDER_KEY, order);
  return new ModelAndView("showOrder", ORDER_KEY, order);
}
catch (NotEnoughInventoryException ex) {
  return new ModelAndView("notEnoughInventory", "exception", ex);
}
}
```

Listing 11-3-13 Creating a New Order

11.3.2.2 Benefit of Proposed WAF and Supporting Infrastructure

A clean web tier, with control flow handled by Java objects and separated from presentation, can be achieved via the proposed Web Application Framework and supporting infrastructure. There is no markup in Java code, and the application built on proposed infrastructure is not bound to a particular view technology.

A thin web tier, with a minimum code volume and the best possible separation of web interface from business logic, can be achieved via the proposed Web Application Framework and supporting infrastructure. The web tier Java code consists of a single class, org.petstore.pet.web.PetController, which is the only class depending on the web framework or Servlet API.

11.4 Summary

The first case study demonstrates the use of OCL as the syntax of formal abstraction rules in the same way that transformation rules are defined in MDA. While this technique is not fully implemented to be used in non-trivial applications, the case study shows a possible way of specifying abstraction rules compatible with MDA.

The next two case studies demonstrate XML transformation and Web Application Infrastructure and Framework. These techniques are practical and can be used to transform a Web-based system to an evolvable one capable of running in a heterogeneous environment and accommodating changes.

CHAPTER 12

Conclusion

While software evolution is a systematic process of reengineering a legacy system to an evolvable system, the technology used in this process must keep evolving as well. At times, a new platform or a new implementing language is required as the targets of applying software reengineering. Could we make our software evolution tools as evolvable as the systems that they are used to create? The answer is "Yes". It was the advent of new technologies and their applications, such as XML and UML, which really transformed not only the way to develop software, but also the way to reengineer a legacy system. The evolvable approach to software evolution is to integrate those traditional methods within those new technologies.

- UML is the de facto technique for modelling and its popularity for a long time stays on representing the idea in a visual way. XML is the de facto technique for data management and information exchange. It has become ubiquitous for all new applications. For existing or legacy applications, XML is the ideal glue to integrate them with new systems. In addition, UML has its XML representation XMI. The two major technologies have been closely integrated.

- Data Mining is another important technology that facilitates building evolvable software systems. They have been heavily used in Web-based applications such search engine. For understanding a complex system with a large number of heterogeneous components, data mining is the most effective way.

- Pattern and Relationship analysis is of significant importance to the evolution of software systems, especially Web-based systems. Still, the difference is the degree of complexity and the magnitude of Web-based systems. Pattern and relationship analysis at different levels and different development phases of a Web-based system can help with a deep understanding of the systems to be built or reengineered.

- Application infrastructure and the Web Application Framework built on it can greatly reduce the dependencies between different layers of a Web-based system. The existing technologies are not well designed and do not meet the requirements for developing evolvable Web-based systems in terms of enforcing a clear separation of different application layers. However the concepts of infrastructure and framework are central to any feasible design of evolvable architecture for Web-based systems.

This research aims to unify the above technologies for a solution to the issues raised in chapter 1 introduction. Each of these technologies is given a detailed discussion on how to apply them in the proposed solution to Web-based Systems Evolution. While some of the technologies (OCL reengineering rules, Web Application Infrastructure and Framework, and XML transformation tool) are examined in the three case studies, others (patterns taxonomy and relationship analysis) can be supported by published studies.

12.1 Summary of Thesis

- Round-Trip Engineering. MDA promotes a unified solution to software development and engineering. However MDA can not achieve its goal on its own. MDA and TDD should be combined together to create an Adaptable MDA to gain the advantages of both, where MDA should be used to create models with project stakeholders to help analyse requirements in architectural and design models and TDD should be used as a critical part of development efforts to ensure clean and working code. An AMDA will facilitate a high-quality, working engineering/reengineering process that meets the actual needs of new or existing systems.

- Web Application Development and Reengineering. A Web Application Infrastructure (WAI) and a Web Application framework (WAF) based on it are presented as an approach to building evolvable Web-based systems. A WAI specifies a collection of built-in system services and configuration mechanisms that provides a running environment for software systems. A good WAI contributes significantly to the success of software development and

reengineering. An evolvable infrastructure eliminates the dependency of applications on it via a Dependency Push container. A good WAF provides a clean and thin Web tier that is extensible and substitutable, allowing for easy and consistent configuration of WAF objects and explicitly separating the roles of controller, model, and view.

- Reverse Engineering. Both formal and cognitive reverse engineering techniques are reviewed in this research. We claim that neither of them by its own can solve the complex issue of Web-based systems evolution. Therefore, both techniques are discussed and incorporated into the whole proposed approach. For formal method based reverse engineering, a set of formalised abstraction rules is introduced to handle transformations at different abstraction levels. For cognitive reverse engineering, a set of formulas for relationship analysis of software systems is given to collect information from Web-based systems as input of the proposed data mining approach specified by a set of formulas. Both techniques should be applied to understand a large Web-based system effectively.

- XML Transformation. One of the challenges software reengineering faces is handling changes from business requirements and technology development. The advantages of XML in data management help software systems adapt to changes in both technologies and businesses. An XML-based transformation approach is given to transform legacy file formats to and from XML. The XML transformation process is divided into two categories according to whether the legacy format is the source or target format. In either case, the transformations share a common structure. Web-based systems are abundant in various file formats for configurations. A unified XML data management for those files could greatly reduce the complexity of development, maintenance and reengineering.

- UML Visualisation. To represent logic view of the recovered information, an Architecture Description for Software Reengineering (ADSR) is developed, representing the relationships between components and connectors in a software

system. The most important attributes of a component are "interface" and "semantics". In ADSR, the interface of the component is a collection of service ports provided or required, and the semantics of the component can be deduced from the extracted OCL specification. ADSR is implemented via UML via UML profiles.

12.2 Success Criteria Revisited

The research results meet the success criteria given in Chapter 1 as follows:

- Can this approach handle the diversity of Web-based systems? The proposed approach has flexibility and extensibility as the main target of building a unified solution to Web-based systems evolution. First, both formal and cognitive reverse engineering techniques are incorporated in this approach to apply in different scenarios. Second, XML data management is very suitable to handle the diversity of file formats in Web-based systems, where XML itself is playing more and more important role. Finally, UML is the de facto standard for representing design concepts, no matter what the implementing techniques are. The proposed UML-based ADL can hide the diversity of Web-based systems to present a unified view.

- Is the target system (either from development or reengineering) capable of accommodating changes? The target model is built on the Web Application Infrastructure and Web Application Framework, both of which are highly flexible and extensible. The infrastructure supports Dependency Push that decouples business logic from configuration of collaborating objects and resources and any changes made to the business logic will have the minimal impact on the whole application. With MVC architecture, the application can choose different controllers, views and models without affecting other components. In terms of the view implementation, different technologies can be used for the same controller and model.

- Can this approach be integrated with MDA and be applied to Web-based systems? Firstly, OCL is adopted to specify the syntax of the formal abstraction

rules in the same way that transformation rules are defined in MDA. Those rules can be used by MDA compliant modeling tools. Secondly, data mining technique is used to tackle the heterogeneity of Web-based systems by analysing relationships implied by not only application source code and hypermedia files, but configuration files. These two techniques, however, are discussed and analysed in this research to complement the whole picture of the proposed solution. Non-trivial implementation of these theories and experimentation are left for future research.

- Is the approach feasible for realisation? For example, is it possible to build a practical tool based on the approach? Not for all components. Quite a lot attention was paid to the practical part of the approach during development. The XML transformation, Web Application Infrastructure and Framework. The examples and case studies show that the approach is a "practical" one, i.e., feasible for practice, in terms of those components. On the other hand, as stated earlier, this research does not provide non-trivial implementation for formal abstraction and data mining. The discussions of those components, however, are supported by published studies and further research can be carried out upon them.

- Is the approach capable for industrial-scaled systems? The transformation, infrastructure and framework components of this approach are capable for industrial-scaled systems and efficient enough for real practice in modernising input/output files of legacy systems. They have been tested in industrial projects [FimatEcc05] and are based on standard technologies and an approach to iterative Model Driven Development based on MDA and TDD.

All components of the proposed approach shown in Figure 4-1-1 have been discussed thoroughly in Chapters 5 ~ 9.

- Chapter 5 presented the infrastructure and framework for building evolvable Web-based systems.

- Chapter 6 defined twelve rules for extracting formal specifications from legacy applications:
    - Translation rules
    - Elementary abstraction rules
    - Architecture abstraction rules
- Chapter 7 first classified relationships between components of software systems into four categories:
    - Aggregation
    - Association
    - Classification
    - Generalisation

  A set of formulas was then given to be applied on those relationships to produce clusters during architecture recovery.
- Chapter 8 introduced a taxonomy on patterns used for models identification from Web-based systems.
    - Domain Architectural Patterns
    - System Design Patterns
    - Integration Patterns
- Chapter 9 constructed a complete XML transformation mechanism for modernising legacy files/information shared between Web-based systems.

The proposed approach is complete in respect of its design for handling the heterogeneity and complexity of Web-based systems and its implementation for

adopting standard techniques in Web application development. Such design goals and implementation choices are essential for any evolution approaches to succeed in building evolvable Web-based systems. It can be claimed that any non-trivial evolution projects of Web-based systems would have to adopt a similar, if not identical, design and implementation strategy used in this research.

12.3 Applicability of Proposed Approach

The proposed approach can be applied effectively on Web-based systems that have the following features.

- Separate back-end server and template-based applications. Back-end server applications refer to program modules that hold business logic, process data and communicate with other systems. Template-based applications refer to program modules that directly participate in displaying views to users. The existence of both forms of application modules indicates a separation of presentation and business/data layers, though the boundary is not always a clear cut. Formal abstractions can be applied on the back-end server program modules with intensive calculation or control logic to extract concise specifications.

- Rich configuration files. With separate back-end server and template-based applications, the existence of rich configuration files is important for understanding the relationships between the two, such as mappings between a controller and a view. Data mining techniques like proposed in this research can be used to explore such relationships to depict an overall picture of the whole system.

- External communications. Web-based systems having external communications often need to process trade orders, emails or other text files in the formats of CSV, flat file or EDI. Such systems can be XML-enabled to export data in an XML format and import data from an XML format and thus be able to communicate with each other in a common language, which can be validated and transformed to various representations.

The proposed approach addresses issues in handling system evolution involving the features described above. It can therefore provide a unified solution to facilitate the development and maintenance of such a system during its whole life cycle.

12.4 Limitation and Future Work

The research presented in this thesis is not the end of the story. This research focused on establishing a general framework and methodology handling the evolution of Web-based systems rather than targeting a specific application or platform. The proposed approach, however, has its limitation and some of its components have not been implemented in this research.

The focus of this research is on creating evolvable Web-based systems. The core of the proposed approach, the Web Application Infrastructure, helps to achieve this goal by decoupling the dependencies between Web applications and their run-time environment. This infrastructure, however, may impose another kind of burden on developers. The proposed infrastructure encourages externalising dependencies to configuration files, which can be changed without significant impact on any involved components and used by data mining techniques to understand the architecture of the whole application. The configuration files could therefore become very large and unmanageable without a proper visualisation and validation tool.

Further research on the proposed approach will elaborate on the following issues.

- Infrastructure and Framework. For the research, the implementation of the proposed Web Application and Framework is enough to demonstrate the Dependency Push concept. However, for developing full-blown Web applications, they need to be further developed and the limitation of the Web Application Infrastructure discussed above need to be addressed.

- Compiler for Abstraction and transformation rules. Compiler is the core of automation of abstraction and transformation rules. The design and implementation of a complete compiler is beyond the current work but should be a priority for future research.

- IDE support. IDE support is also important to successful automation of the proposed approach. An viable solution to this is develop Plug-ins for mature production level IDEs, such as Eclipse and NetBeans, so that the development and reengineering work can be done in them with the full support of IDE supplied functionalities.

- Relationships Analysis for Data Mining. Although a relationship analysis and a set of data mining formulas are proposed for cognitive reverse engineering, there is not enough experiment done due to limited time of this research. In future work, this should be another priority for data mining techniques becomes increasingly popular in industrial areas and there are plenty of production level tools for testing the proposed approach.

- Grammar descriptors for XML transformation. Although it is enough to demonstrate its powerful potential to handle the diversity of file formats in Web-based systems, the XML transformation implemented in this research is not complete enough for practical applications due to still the limited time and resources of this research. This could be an improvement point in future work.

References

[Alexander79] C. Alexander C, *The Timeless Way of Building*, Oxford University Press, 1979.

[Arnold92] R. Arnold, "Software Re-engineering," IEEE Computer Society Press, ISBN 0-8186-3271-2, 1992.

[Arthur98] L. J. Arthur, *Software Evolution: The Software Maintenance Chanllenge*, John Wiley & Sons, 1988.

[Balmas96] F. Balmas, "Prisme: Formalizing Programming Strategies as A Way to Understand Programs," In Eighth International Conference on Software Engineering and Knowledge Engineering, IEEE Computer Society, 1996.

[Balmas97] F. Balmas, "Toward a Framework for Conceptual and Formal Outlines of Programs," In Fourth Working Conference on Reverse Engineering, IEEE Computer Society, pp. 226 – 235, 1997.

[Baxter97] Ira D. Baxter and M. Mehlich, "Reverse Engineering is Reverse Forward Engineering," In Proceedings of the Fourth IEEE Working Conference on Reverse Engineering, IEEE, 1997.

[Behforooz96] A. Behforooz and F. J. Hudson, *Software Engineering Fundamentals*, Oxford University Press, 1996.

[Bendifallah87] S. Bendifallah and W. Scacchi, "Understanding Software Maintenance Work," IEEE Transactions on Software Engineering SE-13 (3), 311-323, 1987.

[Bowen91] J.P. Bowen, P.T. Breuer and K. Lano, The REDO Project: Final Report, Technical Report PRG-TR-23-91, Oxford University, 1991.

[Breiman84] L. Breiman, J. H. Friedman, R. A. Olshen and C. J. Stone, *Classification and Regression Trees*, Monterey, CA: Wadsworth and Brooks/Cole, 1984.

[Burbeck92] S. Burbeck, "Application Programming in Smalltalk-80: How to use Model-View-Controller (MVC)," University of Illinois in Urbana-Champaign (UIUC) Smalltalk Archive, 1992.

[Carter02] K. Carter, Supporting Model Driven Architecture with eXecutable UML, White Paper CTN 80, v2.2.

[Ceri00] S. Ceri, P. Fraternali and A. Bongio, *Web Modeling Language (WebML): a modeling language for designing Web sites*, Computer Networks, 2000.

[Ceri03] S. Ceri, P. Fraternali, A. Bongio, M. Brambilla, S. Comai and M. Matera, *Designing Data-Intensive Web Applications*, Morgan Kaufmann Publishing, 2003.

[Compuware05] Compuware Corporation: Homepage, Internet site address http://www.compuware.com, 2005.

[Christodoulou98] S. Christodoulou and G. Styliaras, "Papatheodourou, Evaluation of Hypermedia Application Development and Management Systems," Proceedings of ACM Hypertext Conference, Pittsburgh, 1-10, 1998.

[Conallen03] J. Conallen, *Building Web Applications with UML - second edition*, The Addison-Wesley Object Technology Series, 2003.

[Conallen99] J. Conallen, "Modeling Web Application Architectures with UML," Communications of the ACM, vol. 42, issue 10, ACM Press, New York, 1999.

[CS96] P. Cheeseman and J. Stutz, Baysian Classification (autoclass): Theory and Results, In U.M. Fayyad, G. Piatetsky-Shapiro, P. Smith, and R. Uthurusamy, editors, Advances in Knowledge Discovery and Data Mining, pages 153--180. AAAI/MIT Press, 1996.

[Dashofy02] E. Dashofy, A. Hoek and R. Talor, "An Infrastructure for the Rapid Development of XML-based Architecture Description Languages," IEEE International Conference on Software Engineering. Orlando, USA. 266-276, 2002.

[Dijkstra76] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[Dijkstra90] E. W. Dijkstra and C. S. SCHOLTEN, *Predicate Calculus and Program Semantics*, Springer-Verlag, 1990.

[Duffy04] D. Duffy, *Domain Architectures : Models and Architectures for UML Applications*, John Wiley & Sons, 2004.

[EAA03] M. Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003.

[FermaT] http://www.dur.ac.uk/martin.ward/fermat.html.

[Fielding98] R. Fielding, JR. Whitehead, K. Anderson, G. Bolcer, P. Oreizy and R. Taylor, Web-Based Development of Complex Information Products, Communications of the ACM, 41(8), 84-92, 1998.

[FimatEcc05] FimatEcc project, Fimat International Banque, London, 2005.

[Finnigan97] P. Finnigan, R. C. Holt, et al., The Software Bookshelf, IBM Systems Journal, 36(4), 564-593, 1997.

[Fis95] D. Fisher, "Optimization and Simplification of Hierarchical Clusterings," In Proceedings of the First International Conference on Knowledge Discovery and Data Mining, p 118--123, Montreal, Quebec, 1995.

[Frankel03] D. Frankel, *Model Driven Architecture Applying MDA to Enterprise Computing,* John Wiley & Sons, New York, USA, 2003.

[Fraternali00] P. Fraternali and P. Paolini, Model-driven development of Web applications: the AutoWeb system. ACM Transactions on Information Systems, 18(4):323{382, 2000.

[Fuentes02] L. Fuentes, J. Troya and A. Vallecillo, Using UML Profiles for Documenting Web-Based Application Frameworks, Annals of Software Engineering. Kluwer Academic Publishers, Dordrecht, The Netherlands. 13: 249-264, 2002.

[Gamma95] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley. 1995.

[Gannod95] Gerald C. Gannod and Betty H. C. Cheng, Strongest Postcondition as the Formal Basis for Reverse Engineering, Journal of Automated Software Engineering, 3(1/2):139-164, June 1996.

[Gannod99] G. Gannod and B. Cheng, "A Framework for Classifying and Comparing Software Reverse Engineering and Design Recovery Techniques," Proceedings of the 6th Working Conference on Reverse Engineering, October 1999.

[Garlan00] D. Garlan, S. Cheng and A. Kompanek, Reconciling the Needs of Architectural Description with Object-Modeling Notations, Science of Computer Programming. Elsevier North-Holland, Inc., Amsterdam, The Netherlands. 44: 23-49, 2000.

[Gerald20] M. Ward, F.W. Calliss and M. Munro, "The Maintainer's Assistant," In Proceedings for the Conference on Software Maintenance, IEEE, 1989.

[Guo99] G. Guo, J. Atlee and R. Kazman, "A Software Architecture Reconstruction Method," Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1), San Antonio, Texas, February 22-24, pp 225-243, 1999.

[Hassan03] A. Hassan and R. Holt, "A Visual Architectural Approach to Maintaining Web Applications," Annals of Software Engineering, Vol. 16, Special Volume on Software Visualization, 2003.

[Haykin94] S. Haykin, *Neural Networks: A Comprehensive Foundation,* New York: Macmillan College Publishing Co., 1994.

[HCC93] J. Han, Y. Cai and N. Cercone, "Data-driven discovery of quantitative rules in relational databases," In IEEE Transactions on Knowledge and Data Eng., volume 5, pages 29--40, 1993.

[Hennicker00] R. Hennicker and N. Koch, "A UML-based Methodology for Hypermedia Design," In Andy Evans, Stuart Kent, and Bran Selic, editors, UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings, volume 1939 of LNCS, pages 410{424. Springer, 2000.

[Hennicker01] R. Hennicker and N. Koch, Systematic design of web applications with UML, In Keng Siau and Terry Halpin, editors, Uni¯ed Modeling Language: Systems Analysis, Design and Development Issues, chapter 1, pages 1{20. Idea Publishing Group, 2001.

[iBatis05] iBATIS, http://www.ibatis.com/.

[IOSoftware05] IO-Software: Homepage, Internet site address http://www.io-software.com Accessed on Apr 10, 2005.

[Isakowitz95] T. Isakowitz, E. Stohr and P. Balasubramanian, A methodology for structuring hypermedia design, Communications of the ACM 38(8), 34-44, 1995.

[ITL] Interval Temporal Logic, http://www.cse.dmu.ac.uk/~cau/itlhomepage/.

[Jacobson91] I. Jacobson and F. LindstrÄom, "Re-engineering of Old System to An Object-Oriented Architecture," Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOP-SLA'91), Phoenix, Arizona, 340-350, 1991.

[KABC96] R. Kazman, G. Abowd, L. Bass and P. Clements, Scenario Based Analysis of Software Architecture. IEEE Software, pages 47{55, November 1996.

[Kazdin03] A. E. Kazdin, Methodological issues and strategies in clinical research (3rd ed., pp. 5–22), Washington, DC: American Psychological Association, 2003.

[KC99] R. Kazman and S. Carrière, "Playing Detective: Reconstructing Software Architecture from Available Evidence," Journal of Automated Software Engineering 6(2), April 1999.

[Kleppe03] A. Kleppe, J. Warmer and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise,* Addison Wesley, Boston, USA, 2003.

[Koufaris98] M. Koufaris, "Structured design of WWW and Intranet applications," 7th International WWW Conference Tutorial, 1998.

[KR90] L. Kaufman and P.J. Rousseeuw, *Finding Groups in Data: an Introduction to Cluster Analysis,* John Wiley & Sons, 1990.

[Kri97] R.L. Krikhaar, "Reverse Architecting Approach for Complex Systems," In Proceedings International Conference on Software Maintenance, pages 4{11. IEEE Computer Society, 1997.

[Kru95] P. Kruchten, The 4 + 1 View Model of Architecture. IEEE Software, pages 42{50, November 1995.

[Lange94] D. Lange, "An Object-Oriented Design Method for Hypermedia Information Systems," Proceedings of the Twenty-seventh Annual Hawaii International Conference on System Sciences, 366-375, 1994.

[Lanham01] T. Lanham, "Designing innovative enterprise portals and implementing them into your content strategies: Lockheed Martin's compelling case study," Proceedings from the Web Content II: Leveraging Best-of-Breed Content Strategies meeting, San Francisco.

[Larman04] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, Third Edition*, Addison Wesley Professional, 2004.

[Larose04] D. T. Larose, *Discovering Knowledge in Data: An Introduction to Data Mining*, Wiley-Interscience, 2004.

[leh85] Lehman MM and Belady LA, *Program Evolution -Process of Software Change*, Acad. Press, London, 1985.

[Li02] Y. Li, Automating Domain Knowledge Recovery from Legacy Software Code, Ph.D. Thesis, De Montfort Univeristy Software Technology Research Laboratory, 2002.

[Liu99] X. Liu, Abstraction: A Notion for Reverse Engineering, PhD thesis, De Montfort University, September, 1999.

[Lowe99] D. Lowe and W. Hall, *Hypermedia & the web: An Engineering Approach,* John Wiley & Sons, 1999.

[MAR96] M. Mehta, R. Agrawal and J. Rissanen, "SLIQ: A Fast Scalable Classifier for Data Mining," In Proceedings of the Fifth International Conference on Extending Database Technology, Avignon, France, 1996.

[Marczyk05] G. Marczyk, D. DeMatteo and D. Festinger, *Essentials of Research Design and Methodology*, ISBN: 0471470538, John Wiley & Sons, 2005.

[Mayrhauser92] von Mayrhauser, Anniliese and Vans, Marie, An Industrial Experience With an Integrated Code Comprehension Model (Technical Report CS-92-205). Ft. Collins, CO: Colorado State University, 1992.

[Mayrhauser95] von Mayrhauser, Anniliese and Vans, Marie, "Program Comprehension During Software Maintenance and Evolution," Computer 28, 8: 44-55, 1995.

[Medvidovic2000] N. Medvidovic and R. Taylor, A Classification and Comparison Framework for Software Architecture Description Languages, IEEE Transactions on Software Engineering, vol. 26, no. 1, pp. 70-93, January 2000.

[Mellor02] S. J. Mellor and M. J. Balcer, *Executable UML: A Foundation for Model-Driven Architecture,* Boston: Addison Wesley, 2002.

[Merialdo03] P. Merialdo, P. Atzeni and G. Mecca, Design and development of data-intensive web sites: The Araneus approach, ACM Transactions on Internet Technology (TOIT), 3(1):49{92, 2003.

[Micro04] *Enterprise Solution Patterns Using Microsoft .NET*, Microsoft Press. 2004.

[morgan72] R. Duda and P. Hart, *Pattern Classification and Scene Analysis*, JohnWiley & Sons, 1973.

[Moszkowski85] B. Moszkowski, A Temporal Logic for Multilevel Reasoning about Hardware. IEEE Computer Society, 1985.

[Moszkowski86] B. Moszkowski, Executing Temporal Logic Programs, Cambridge University Press, Cambridge UK, 1986.

[Netbeans05] http://www.netbeans.org/, 2005.

[NH94] R. Ng and J. Han, "Efficient and Effective Clustering Method for Spatial Data Mining," In Proceedings of the 20th VLDB Conference, pages 144--155, Santiago, Chile, 1994.

[Pen92] D. Penny, The Software Landscape: A Visual Formalism for Programming-in-the-Large. PhD thesis, University of Toronto, 1992.

[Peritus] Peritus Software Services, http://www.peritus.com/.

[Pressman94] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, McGraw-Hill, 1994.

[Quinlan86] J. R. Quinlan, "Induction on Decision Trees," Machine Learning, vol. 1, pp. 81-106, 1986.

[Rawlins03] M. Rawlins, *Using XML with Legacy Business Applications*, ISBN: 0321154940; Published: Aug 6, 2003.

[Rayside00] D. Rayside, S. Reuss, E. Hedges and K. Kontogiannis, "The Effect of Call Graphs Construction Algorithms for Object-oriented Programs on Automatic Clustering," In Proceedings of the 8th International Workshop on Program Comprehension. IEEE, 2000.

[Ricca03] F. Ricca and P. Tonella, "Using Clustering to Support the Migration from Static to Dynamic Web Pages," 11th International Workshop on Program Comprehension, pp. 207-216, Portland, Oregon, USA, May 2003.

[Scholefield92] D. Scholefield, A refinement Calculus for Real-time Systems. PhD thesis (1992).

[Schull96] F. Schull, W. Melo, V. Basili, An Inductive Method for Discovering Design Patterns from Object-Oriented Software Systems, UMIACS-TR-96-10, University of Maryland, 1996.

[Schwabe96] D. Schwabe, G. Rossi and S. Barbosa, "Systematic Hypermedia Application Design with OOHDM," Proceedings of Hypertext, Washington DC, pp 116-128, 1996.

[Seacord03] R. Seacord, D. Plakosh and G. Lewis, *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*, Addison Wesley, 2003.

[shaw95] M. Shaw and D. Garlan, *Software Architecture,* Prentice-Hall, 1995.

[Sneed88] H. M. Sneed and G. Jandrasics, "Inverse Transformation of Software from Code to Specification," In IEEE Conference on Software Maintenance, 1988.

[Sneed95] H. M. Sneed, Planning the Reengineering of Legacy Systems, IEEE Software 12, 1 (January 1995).

[Spring04] N. Spring, D. Wetherall, T. Anderson, ACM SIGCOMM Computer Communication Review, vol. 34, issue 1, 3-8, ACM Press, USA, 2004.

[Storey97] D. Storey, D. Fracchia and H. Müller, "Cognitive Design Elements to Support the Construction of a Mental Model During Software Visualization," 17-28. Proceedings of the 5th Workshop on Program Comprehension. Dearborn, Michigan: May 28-30, 1997. Los Alamitos, CA: IEEE Computer Society Press, 1997.

[struts05] Apache Struts, http://struts.apache.org/.

[Swanson76] E. B. Swanson, "The dimensions of maintenance," Proceedings of the 2nd International Conference on Software Engineering (ICSE'76), San Francisco, California, 492{497, 1976.

[Synytskyy03] M. Synytskyy, J. R. Cordy, T. R. Dean, "Resolution of Static Clones in Dynamic Web Pages," 5th International Workshop on Web site Evolution, pp. 49-58, Amsterdam, Netherlands, September 2003.

[Tonella03] P. Tonella, F. Ricca, E. Pianta and C. Girardi, "Using Keyword Extraction for Web site Clustering," Proceedings of WSE 2003, 5th International Workshop on Web site Evolution, pp. 41-48, Amsterdam, The Netherlands, September 22, 2003.

[Tou74] J. T. Tou and R. C. Gonzalez, *Pattern Recognition Principles*. London: Addison-Wesley, 1974.

[Velocity] http://jakarta.apache.org/velocity/.

[W3C03] Scalable Vector Graphics (SVG) 1.1 Specification, W3C Recommendation, http://www.w3.org/TR/SVG/, 2003.

[Waldo94] J. Waldo, G. Wyant, A. Wollrath and S. Kendall, "A Note on Distributed Computing," Technical Report SMLI TR-94-29, Sun Microsystems Laboratories, November 1994.

[Ward89] M. Ward, F.W. Calliss and M. Munro, "The Maintainer's Assistant," In Proceedings for the Conference on Software Maintenance. IEEE, 1989.

[Warmer98] J. Warmer and A. Kleppe, *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, 1998.

[warren99] I. Warren, *The Renaissance of Legacy Systems, Method Support for Software-System Evolution*, Springer, 1999.

[Wikipedia05] Formal Methods, http://en.wikipedia.org/wiki/Formal_method.

[WK91] S.M. Weiss and C. A. Kulikowski, *Computer Systems that Learn: Classification and Prediction Methods from Statistics, Neural Nets, Machine Learning, and Expert Systems,* Morgan Kaufmann, San Mateo, CA, 1991.

[Yang91] H. Yang, "The supporting environment for A Reverse Engineering System - the Maintainer's Assistant," IEEE Conference on Software Maintenance-1991, pp. 13–22, 1991.

[Yang94] H. Yang, Acquiring Data Designs from Existing Data Intensive Programs, Ph.D. thesis, Durham University, 1994.

[Yang98] H. Yang, X. Liu and H. Zedan, "Tackling the Abstraction Problem for Reverse Engineering in a System Re-engineering Approach," In the proceedings of the IEEE Conference on Software Maintenance (ICSM'98), IEEE Computer Society, 1998.

[Yang03] H. Yang and M. Ward, *Successful Evolution of Software Systems*, Artech House Publishers, 2003.

[Yoo04] J. Yoo, J. Catanio, M. Bieber and R. Paul, "Relationship Analysis in Requirements Engineering," Requirements Engineering Journal, Springer Verlag London Ltd., October, 2004.

APPENDIX A UML Profile for Web Application Extensions

Type	Stereotype	Description
Class	ServerPage	A server page is a Web page that has scripts executed by the server. A server page can only interact with objects on the server. A tagged value for a server page indicates its implementing language.
Class	Client Page	A client page is a simple HTML Web page that contains data and presentation, and any local scripts. Three tagged values TitleTag (the title of the page), BaseTag (the base URL) and BodyTag (the HTML <body> tag that sets fonts and background colors) are defined for client page.
Class	Form	A form is an object with a set of fields to be submitted by users. The attributes of a «form» class are the names of the fields. A «form» class has no operations. Its attributes are manipulated by operations belonging to the page on which the form resides.
Class	Frameset	A frameset comprises a group of Web pages grouped in a frame. Two tagged values are defined for frameset: rows, which is a string of comma-separated values of row heights, and cols, which is a string of comma-separated column widths.
Class	Target	A target is a window in which a web page can be displayed.
Class	JavaScript	JavaScript objects are represented by a class of stereotype «JavaScript Object».
Class	ClientScript Object	A ClientScript Object is a collection of client-side scripts placed in a separate file and invoked by a separate request from the browser.
Component	Web Page	A Web Page is an HTML-formatted page that contains HTML text code and sometimes scripts that run on the client browser, and/or on the server. It may also contain a compiled module that loads on the server. The full URL or path of the Web Page is specified as a tagged value.
Component	ASP Page	An Active Server Page (ASP) Page implements ASP code on the server. The full URL or path of the ASP Page is specified as a tagged value.
Component	JSP Page	A Java Server Page (JSP) Page implements JSP code on the server. The full URL or path of the JSP Page is specified as a tagged value.
Component	Servlet	A Servlet is a Java component that runs on the server. The

		full URL or path of the Servlet is specified as a tagged value.
Component	Script Library	A Script Library is a component that provides a library of functions and subroutines used by other components. The full URL or path of the script library is specified as a tagged value.

Table A-1 UML Profile for Web Application Extensions

APPENDIX B Grammar Description Schemas

B.1 CSV Source Grammar Description Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   elementFormDefault="unqualified"
   attributeFormDefault="unqualified">
<xs:include schemaLocation="CSVCommonGrammarDescription.xsd"/>
<xs:element name="CSVSourceGrammarDescription">
 <xs:annotation>
  <xs:documentation>
   This schema specifies the format of Grammar Description
   Documents when converting from CSV files as source to XML
   documents as targets
  </xs:documentation>
 </xs:annotation>
 <xs:complexType mixed="false">
  <xs:sequence>
   <xs:element name="PhysicalCharacteristics"
      type="CSVPhysicalCharacteristicsType"/>
   <xs:element name="XMLOutputCharacteristics"
      type="CSVXMLOutputCharacteristicsType"/>
   <xs:element name="Grammar"
      type="CSVGrammarType"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
</xs:schema>
```

Listing B-1 CSVSourceGrammarDescription.xsd

B.2 CSV Target Grammar Description Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   elementFormDefault="unqualified"
   attributeFormDefault="unqualified">
 <xs:include schemaLocation="CSVCommonGrammarDescription.xsd"/>
 <xs:element name="CSVTargetGrammarDescription">
<xs:annotation>
 <xs:documentation>
  This schema specifies the format of Grammar Description
  Documents when converting from XML documents as source to
  CSV files as targets
 </xs:documentation>
</xs:annotation>
```

```
  <xs:complexType mixed="false">
   <xs:sequence>
    <xs:element name="PhysicalCharacteristics"
      type="CSVPhysicalCharacteristicsType"/>
    <xs:element name="Grammar"
      type="CSVGrammarType"/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Listing B-2 CSVTargetGrammarDescription.xsd

B.3 CSV Grammar Description Common Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="unqualified"
  attributeFormDefault="unqualified">
 <xs:include schemaLocation="CommonGrammarDescription.xsd"/>
 <xs:complexType name="CSVPhysicalCharacteristicsType"
   mixed="false">
  <xs:annotation>
   <xs:documentation>
     Describes the CSV physical record organization
   </xs:documentation>
  </xs:annotation>
  <xs:sequence>
   <xs:element name="RecordTerminator">
    <xs:complexType>
     <xs:complexContent>
      <xs:extension base="EmptyType">
       <xs:attribute name="value"
         type="RecordTerminatorValueType"
         use="required"/>
      </xs:extension>
     </xs:complexContent>
    </xs:complexType>
   </xs:element>
   <xs:element name="ColumnDelimiter" type="DelimiterType"/>
   <xs:element name="TextDelimiter" type="DelimiterType"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="CSVXMLOutputCharacteristicsType"
  mixed="false">
 <xs:annotation>
  <xs:documentation>
   Describes characteristics of the output XML document
  </xs:documentation>
</xs:annotation>
<xs:sequence>
 <xs:element name="DocumentBreakColumn">
  <xs:complexType mixed="false">
```

```
    <xs:complexContent mixed="false">
     <xs:extension base="EmptyType">
      <xs:attribute name="value" type="BreakColumnType"
        use="required"/>
     </xs:extension>
    </xs:complexContent>
   </xs:complexType>
  </xs:element>
  <xs:element name="PartnerBreakColumn">
   <xs:complexType mixed="false">
    <xs:complexContent mixed="false">
     <xs:extension base="EmptyType">
      <xs:attribute name="value" type="BreakColumnType"
        use="required"/>
     </xs:extension>
    </xs:complexContent>
   </xs:complexType>
  </xs:element>
  <xs:element name="SchemaLocationURL" minOccurs="0">
   <xs:complexType>
    <xs:complexContent>
     <xs:extension base="EmptyType">
      <xs:attribute name="value" type="anyURI127"
        use="required"/>
     </xs:extension>
    </xs:complexContent>
   </xs:complexType>
  </xs:element>
 </xs:sequence>
</xs:complexType>
<xs:complexType name="CSVGrammarType">
 <xs:annotation>
  <xs:documentation>
   Describes the grammar of the CSV file
  </xs:documentation>
 </xs:annotation>
 <xs:sequence>
  <xs:element name="RowDescription">
   <xs:annotation>
    <xs:documentation>
     Describes a row in the CSV file. Currently, all rows
     must have the same format so we only allow a single
     one of these Elements.
    </xs:documentation>
   </xs:annotation>
   <xs:complexType>
    <xs:sequence>
     <xs:element name="ColumnDescription" maxOccurs="100">
      <xs:annotation>
       <xs:documentation>
        Describes a column in the row. The current
        design limits us to one hundred columns per
        row.
       </xs:documentation>
```

```
    </xs:annotation>
    <xs:complexType>
     <xs:complexContent>
      <xs:extension base="FieldGrammarType">
       <xs:attribute name="DelimitText"
          type="xs:boolean" use="optional"
          default="false"/>
      </xs:extension>
     </xs:complexContent>
    </xs:complexType>
   </xs:element>
</xs:sequence>
<xs:attribute name="ElementName" type="NMToken127"
     use="required"/>
    </xs:complexType>
   </xs:element>
 </xs:sequence>
 <xs:attribute name="ElementName" type="NMToken127"
    use="required"/>
</xs:complexType>
<xs:simpleType name="BreakColumnType">
 <xs:annotation>
  <xs:documentation>
   Enforces restrictions on column number for partner and
   document break
  </xs:documentation>
 </xs:annotation>
 <xs:restriction base="xs:nonNegativeInteger">
  <xs:maxExclusive value="100"/>
 </xs:restriction>
</xs:simpleType>
</xs:schema>
```

Listing B-3 CSVCommonGrammarDescription.xsd

APPENDIX C Web Application Infrastructure and Framework

C.1 Web Application Infrastructure

C.1.1 Design Issues of Evolvable Applications

C.1.1.1 Template Method Design Pattern

An order processing system might involve price calculation, customer credit check and available discount. It is desirable to separate the steps of the business logic from underlying persistence system, such as a RDBM.

The AbstractOrderEJB superclass implements the business logic for checking customer credit limit and applying discount to certain orders. Its public checkoutOrder() method, shown in Listing C -1-1, is final so that this workflow can not be modified by subclasses.

```
public final Invoice checkoutOrder (int customerId, InvoiceItem[] items)
   throws NoSuchCustomerException, CreditLimitViolation {

 int total = 0;
 for (int i = 0; i < items. length; i++) {
    total += getItemPrice (items [i]) * items [i] .getQuantity();
 }

 if (total > getCreditLimit (customerId) ){
    getSessionContext() .setRollbackOnly();
    throw new CreditLimitViolation (total, limit);
 }
 else if (total > DISCOUNT_THRESHOLD) {
    // Apply discount to total...
 }

    int invoiceId = checkoutOrder (customerId, total, items);
    return new InvoiceImpl (iid, total);
}
```

Listing C-1-1 Example of Template Method

Steps of a workflow are defined as protected abstract "template methods", shown in Listing C-1-2, which must be implemented by subclasses.

```
protected abstract int getItemPrice(InvoiceItem item);

protected abstract int getCreditLimit(customerId)
  throws NoSuchCustomerException;

protected abstract int checkoutOrder(int customerId, int total,
        InvoiceItem[] items);
```

Listing C-1-2 Template Methods for Workflow Steps

Template Method pattern enables good separation of concerns, where the superclass specify the business logic, while the subclasses concentrate on implementing low-level operations, such as JDBC. As the template methods are protected, the details of the class's implementation are hidden from callers..

C.1.1.2 Strategy Design Pattern

Strategy design pattern can be used to achieve similar result as Template design pattern. It defines an interface for the "Template methods" as shown in Listing C-1-3.

```
public interface OrderHelper {
   int getItemPrice (InvoiceItem item);
   int getCreditLimit (customerId) throws NoSuchCustomerException;
   int checkoutOrder (int customerId, int total, InvoiceItem[] items);
}
```

Listing C-1-3 Interface for Strategy Design Pattern

A concrete OrderEJB class depending on an instance variable of this interface can set this helper object via constructor or a bean property as shown in Listing C-1-4.

```
private OrderHelper dataHelper;

public void setOrderHelper (OrderHelper oataHelper) {
   this.orderHelper = orderHelper;
}
```

Listing C-1-4 Bean Property for Setting Helper Object

The implementation of the checkoutOrder() method is similar to that for the Template Method pattern, except that it invokes the operations on the instance of the helper interface as shown in Listing C-1-5.

```
public final Invoice checkoutOrder (int customerId, InvoiceItem[] items)
    throws NoSuchCustomerException, CreditLimitViolation {

 int total = 0;
 for (int i = 0; i < items.length; i++) {
  total += this.orderHelper.getItemPrice(items[i]) *
   items[i].getQuantity();
 }

 if (total > this.orderHelper.getCreditLimit(customerId)) {
  getSessionContext() .setRollbackOnly();
  throw new CreditLimitViolation(total, limit);
 } else if (total > DISCOUNT_THRESHOLD) {
  // Apply discount to total...
 }

 int invoiceId = this.orderHelper.checkoutOrder (customerId, total, items);
 return new InvoiceImpl (iid, total);
}
```

Listing C-1-5 Example of Strategy Design Pattern

Strategy pattern is more complex than the Template Method pattern, but is more flexible.

C.1.1.3 Using Callbacks to Achieve Extensibility

Callback pattern is effective when working with low-level APIs such as JDBC. A simplified JDBC utility class could use a query() method that takes as parameters a SQL query string and an implementation of a callback interface, shown in Listing C-1-6, to be invoked for each row of the result set the query generates.

```
public interface RowCallbackHandler {
 void processRow(ResultSet rs) throws SQLException;
}
```

Listing C-1-6 RowCallbackHandler Interface

The query() method separates calling code from the details of retrieving a JDBC connection, creating and using a statement, releasing resources, and handling errors, as shown in Listing C-1-7.

```
public void query(String sql, RowCallbackHandler callbackHandler)
   throws JdbcSqlException {

  Connection con = null;
  PreparedStatement ps = null;
  ResultSet rs = null;
  try {
   con = <code to get connection>
   ps = con.prepareStatement (sql);
   rs = ps.executeQuery();
   while (rs.next()) {
    callbackHandler.processRow(rs);
   }
      rs.close();
      ps.close();
   } catch (SQLException ex) {
      ...
   }
   finally {
      DataSourceUtils.closeConnectionIfNecessary(this.dataSource, con);
   }
  }
```

Listing C-1-7 Callback Pattern to Hide Implemenation Details

C.1.1.4 Observer Design Pattern to Decouple Components

The following code comes from the GUI component of the XMLTransformer. It intercepts property change events and sets corresponding text area.

```
   public void propertyChange(PropertyChangeEvent e) {
      if (e.getSource() instanceof org.evolvable.xmltransformer.transform.CSVSourceTransformer) {
         jTextArea1.setText(jTextArea1.getText() + "\n" + e.getNewValue().toString());
      }

      if (e.getSource() instanceof org.evolvable.xmltransformer.transform.CSVTargetTransformer) {
         jTextArea2.setText(jTextArea2.getText() + "\n" + e.getNewValue().toString());
      }
}
```

Listing C-1-8 Observer Design Pattern in GUI Component

C.1.2 Enhanced Bean-based Manipulation

The lowest layer of the proposed infrastructure is the org.evolvable.beans package, which provides extra ability to manipulate JavaBeans.

C.1.2.1 BeanHandler Interface and BeanHandlerImpl

The most important methods in the BeanHandler interface include:

```
Object getPropertyValue (String propertyName) throws BeansException;
```

This returns a property value, given a property name. The string "order" would return a property exposed via a getOrder() accessor method.

The setPropertyValue() method is a parallel setter method with the following signature:

```
void setPropertyValue (String propertyName, Object value)
    throws PropertyVetoException, BeansException;
```

This method throws the java.beans.PropertyVetoException to allow for event listeners to "veto" property changes. Another setter method has the following signature:

```
void setPropertyValue (PropertyValue pv)
    throws PropertyVetoException, BeansException;
```

This method takes a simple object holding the property name and value to allow combined property updates via the following method:

```
void setPropertyValues (PropertyValues pvs) throws BeansException;
```

This sets a number of properties in a single update, which is useful in situations, such as when populating JavaBeans from HTTP request parameters.

TimeBean is a concrete class implementing the interface ITimeBean that contains four properties: hour(int), minute(int), second(int) and time24 (ITimeBean) as shown in Listing C-1-9.

```
public interface ITimeBean {
  int getHour();
  void sethour(int hour);
  int getMinute();
  void setMinute(int minute);
  int getSecond();
  void setSecond(int second);
  ITimeBean getTime24();
  void setTime24(ITimeBean time24);
  ITimeBean getTime12();
  void setTime12(ITimeBean time12);
  void setTimeFormat(int timeFormt);
  int getTimeFormat();
}
```

Listing C-1-9 TimeBean Definition

Using BeanHandler object, the property values of TimeBean can be easily manipulated, as shown in Listing C-1-10.

```
TimeBean starttime = new TimeBean ();
BeanHandler bw = new BeanHandlerImpl(starttime);
bw.setPropertyValue("hour", new Integer(32));
bw.setPropertyValue("minute", new Integer(10));
bw.setPropertyValue("time24", new TimeBean());
bw.setPropertyValue("time24.minute", new Integer(5));
bw.setPropertyValue("time24.time24", starttime);
bw.setPropertyValue("timeFormat", new Integer(12));
Integer hour = (Integer) bw.getPropertyValue("hour");
Integer minute = (String) bw.getPropertyValue("minute");
Integer time24Minute = (Integer) bw.getPropertyValue("time24.minute");
```

Listing C-1-10 Usage of BeanHandler

C.2 Web Application Framework

C.2.1 A Basic Controller Implementation

A simple implementation of the Controller interface is shown in Listing C-2-1, where the controller chooses a view according to the presence and validity of a "view" request parameter.

```
package simple;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class SimpleController implements Controller {

 public ModelAndView processRequest (HttpServletRequest request,
   HttpServletResponse response) throws ServletException {

  String view = request.getParameter ("view") ;
  if (view == null || "" .equals (view) ) {
   return new ModelAndView ("absentView") ; // No name supplied
  } else if (view.indexOf ("-") !=  -1) {
   return new ModelAndView("invalidView", "view", view) ; // Name present but invalid
  } else {
   return new ModelAndView("validView", "view", view) ; // Name present and valid
  }
 }
}
```

Listing C-2-1 A Simple Controller Implementation

C.2.2 A Controller Exposing Bean Properties

As JavaBeans obtained from the application bean factory, controllers are very easy to parameterise.

The following simple controller uses a bean property view set outside Java code to modify the model data output by the controller. This controller directs the user to the absentView if there is not a name parameter. Otherwise, the user is directed to a welcomeView showing a greeting string and the value of the view property of the controller instance:

```
package simple;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class SimpleController extends AbstractController {

  private String view;

  public void setView (String view) {
    this.view = view;
  }

protected ModelAndView processRequestInternal (
    HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

  String pname = request.getParameter ("name") ;
  if (pname == null) {
    return new ModelAndView ("absentView") ;
  } else {
    return new ModelAndView ("welcomeView" , "welcome" ,
                    "Hello" + pname + ", this view is " + this.view) ;
  }
}

protected void init() throws ApplicationRegistryException {
  if (this.view == null)
    throw new ApplicationRegistryException (
      "view property must be set on beans of class " + getClass() .getName() ) ;
  }

}
```

Listing C-2-2 A Simple Controller with Exposed Beans

A controller object can be configured as follows.

```
<bean name="simpleController"
    class="simple.SimpleController" >
    <property name="view">The Bean Controller</property>
</bean>
```

Listing C-2-3 Configuration of view Object

The ability to parameterise controllers in a standard way has advantages as follows.

- There is no need for look up code in web tier controllers.
- There is no need for the Singleton design pattern or factory methods.
- Different object instances of the same controller class can be configured to meet different requirements.
- It's easy to test controller beans outside a Web container.

C.2.3 A Method Mapping Controller

A controller mapped onto one request type is not appropriate in all cases.

- For a large application having a number of request types requiring different business operations with various levels of complexity, there is likely to be a proliferation of tiny controller classes.
- Sometimes many controllers share many configuration properties, and, although concrete inheritance can be used to allow the common properties to be inherited, modeling the controllers as distinct objects is illogical.

A method mapping controller is implemented in proposed framework to address above issues by mapping each request onto the name of a method of a controller class extending MethodMappingControler. In the proposed framework, public request handling methods accepts at least HttpServletRequest and HttpServletResponse objects as parameters and returns a ModelAndView object. Listing C-2-4 shows the signature of a request handling method.

```
public ModelAndView meaningfulMethodName (
   HttpServletRequest request, HttpServletResponse response) ;
```

Listing C-2-4 Request Handling Method Signature

The MethodMappingController factors the logic of choosing request handling method into a simple interface shown in Listing C-2-5 via the Strategy design pattern. An object implementing this interface can be set as a bean property on any method mapping controolers.

```
public interface MethodMapper {
   String getHandlerMethodName (HttpServletRequest request)
   throws NoSuchRequestHandlingMethodException;
}
```

Listing C-2-5 MethodMapper Interface

- The default implementation of MethodMapper, ParameterMethodMapper, maps the method name onto the request URL that contains an action parameter representing the method name.

- The PropertiesMethodMapper implementation allows specifying mappings to be specified in the controller servlet's XML configuration file.

The Method Mapping approach limits the number of controller classes, which may improve maintainability. It is a better model for some use cases. However, unrelated functionality without sharing the same configuration should not be built into a single Method Mapping controller.

APPENDIX D Relationship Analysis of Web-based Systems

D.1 Relationships in Struts Configuration Files

D.1.1 Struts Configuration File

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE struts-config PUBLIC
      "-//Apache Software Foundation//DTD Struts Configuration 1.0//EN"
      "http://jakarta.apache.org/struts/dtds/struts-config_1_0.dtd">

<struts-config>
  <form-beans>
    <form-bean name="accountForm"
          type="org.ibatis.jpetstore.presentation.form.AccountForm"/>
    <form-bean name="cartForm"
          type="org.ibatis.jpetstore.presentation.form.CartForm"/>
    <form-bean name="categoryForm"
          type="org.ibatis.jpetstore.presentation.form.CategoryForm"/>
    <form-bean name="emptyForm"
          type="org.ibatis.jpetstore.presentation.form.EmptyForm"/>
    <form-bean name="itemForm"
          type="org.ibatis.jpetstore.presentation.form.ItemForm"/>
    <form-bean name="productForm"
          type="org.ibatis.jpetstore.presentation.form.ProductForm"/>
    <form-bean name="workingAccountForm"
          type="org.ibatis.jpetstore.presentation.form.AccountForm"/>
    <form-bean name="workingOrderForm"
          type="org.ibatis.jpetstore.presentation.form.OrderForm"/>
  </form-beans>
  <global-forwards>
    <forward name="failure" path="/jsp/Error.jsp" redirect="false"/>
    <forward name="global-orderform" path="/shop/newOrderForm.do"/>
    <forward name="global-signon" path="/jsp/SignonForm.jsp"/>
    <forward name="unknown-error" path="/jsp/Error.jsp"/>
  </global-forwards>
  <action-mappings>
    <action name="cartForm" path="/shop/addItemToCart" scope="session"
          type="org.ibatis.jpetstore.presentation.action.AddItemToCartAction"
          validate="false">
      <forward name="success" path="/jsp/Cart.jsp"/>
    </action>
    <action path="/shop/checkout"
          type="org.ibatis.jpetstore.presentation.action.DoNothingAction"
```

```
      validate="false">
    <forward name="success" path="/jsp/Checkout.jsp"/>
</action>
<action input="/jsp/EditAccountForm.jsp" name="workingAccountForm"
      path="/shop/editAccount" scope="session"
      type="org.ibatis.jpetstore.presentation.action.EditAccountAction"
      validate="true">
    <forward name="success" path="/shop/index.do"/>
</action>
<action name="workingAccountForm" path="/shop/editAccountForm"
      scope="session"
      type="org.ibatis.jpetstore.presentation.action.EditAccountFormAction"
      validate="false">
    <forward name="success" path="/jsp/EditAccountForm.jsp"/>
</action>
<action path="/shop/index"
      type="org.ibatis.jpetstore.presentation.action.IndexAction"
      validate="false">
    <forward name="success" path="/jsp/index.jsp"/>
</action>
<action name="accountForm" path="/shop/listOrders" scope="session"
      type="org.ibatis.jpetstore.presentation.action.ListOrdersAction"
      validate="false">
    <forward name="success" path="/jsp/ListOrders.jsp"/>
</action>
<action input="/jsp/NewAccountForm.jsp" name="workingAccountForm"
      path="/shop/newAccount" scope="session"
      type="org.ibatis.jpetstore.presentation.action.NewAccountAction"
      validate="true">
    <forward name="success" path="/shop/index.do"/>
</action>
<action name="workingAccountForm" path="/shop/newAccountForm"
      scope="session"
      type="org.ibatis.jpetstore.presentation.action.NewAccountFormAction"
      validate="false">
    <forward name="success" path="/jsp/NewAccountForm.jsp"/>
</action>
<action input="/jsp/NewOrderForm.jsp" name="workingOrderForm"
      path="/shop/newOrder" scope="session"
      type="org.ibatis.jpetstore.presentation.action.NewOrderAction"
      validate="true">
    <forward name="confirm" path="/jsp/ConfirmOrder.jsp"/>
    <forward name="shipping" path="/jsp/ShippingForm.jsp"/>
    <forward name="success" path="/jsp/ViewOrder.jsp"/>
</action>
<action name="workingOrderForm" path="/shop/newOrderForm"
      scope="session"
      type="org.ibatis.jpetstore.presentation.action.NewOrderFormAction"
      validate="false">
    <forward name="success" path="/jsp/NewOrderForm.jsp"/>
</action>
<action name="cartForm" path="/shop/removeItemFromCart" scope="session"
      type="org.ibatis.jpetstore.presentation.action.RemoveItemFromCartAction"
      validate="false">
```

```
    <forward name="success" path="/jsp/Cart.jsp"/>
</action>
<action name="emptyForm" path="/shop/searchProducts"
        type="org.ibatis.jpetstore.presentation.action.SearchProductsAction"
        unknown="false" validate="false">
    <forward name="success" path="/jsp/SearchProducts.jsp"/>
</action>
<action path="/shop/signoff"
        type="org.ibatis.jpetstore.presentation.action.SignoffAction"
        validate="false">
    <forward name="success" path="/shop/index.do"/>
</action>
<action name="accountForm" path="/shop/signon" scope="session"
        type="org.ibatis.jpetstore.presentation.action.SignonAction"
        validate="false">
    <forward name="success" path="/shop/index.do"/>
</action>
<action path="/shop/signonForm"
        type="org.ibatis.jpetstore.presentation.action.DoNothingAction"
        validate="false">
    <forward name="success" path="/jsp/SignonForm.jsp"/>
</action>
<action name="cartForm" path="/shop/updateCartQuantities"
        scope="session"
        type="org.ibatis.jpetstore.presentation.action.UpdateCartQuantitiesAction"
        validate="false">
    <forward name="success" path="/jsp/Cart.jsp"/>
</action>
<action name="cartForm" path="/shop/viewCart" scope="session"
        type="org.ibatis.jpetstore.presentation.action.DoNothingAction"
        validate="false">
    <forward name="success" path="/jsp/Cart.jsp"/>
</action>
<action input="/jsp/index.jsp" name="categoryForm"
        path="/shop/viewCategory"
        type="org.ibatis.jpetstore.presentation.action.ViewCategoryAction"
        validate="true">
    <forward name="success" path="/jsp/Category.jsp"/>
</action>
<action input="/jsp/Product.jsp" name="itemForm" path="/shop/viewItem"
        type="org.ibatis.jpetstore.presentation.action.ViewItemAction"
        validate="true">
    <forward name="success" path="/jsp/Item.jsp"/>
</action>
<action name="accountForm" path="/shop/viewOrder" scope="session"
        type="org.ibatis.jpetstore.presentation.action.ViewOrderAction"
        validate="false">
    <forward name="success" path="/jsp/ViewOrder.jsp"/>
</action>
<action input="/jsp/index.jsp" name="productForm"
        path="/shop/viewProduct"
        type="org.ibatis.jpetstore.presentation.action.ViewProductAction"
        validate="true">
    <forward name="success" path="/jsp/Product.jsp"/>
```

```
    </action>
  </action-mappings>
</struts-config>
```

Listing D-1-1 Struts Configuration File for Pet Store

D.1.2 Relationships in Struts Configuration File

Path	Class	Form	Forward Action	Forward JSP	Input
/shop/addItemToCart	org.ibatis.jpetstore.presentation.action.AddItemToCartAction	cartForm		/jsp/Cart.jsp	
/shop/checkout	org.ibatis.jpetstore.presentation.action.DoNothingAction			/jsp/Checkout.jsp	
/shop/editAccount	org.ibatis.jpetstore.presentation.action.EditAccountAction	workingAccountForm	/shop/index.do (org.ibatis.jpetstore.presentation.action.IndexAction)		/jsp/EditAccountForm.jsp
/shop/editAccountForm	org.ibatis.jpetstore.presentation.action.EditAccountFormActi	workingAccountForm		/jsp/EditAccountForm.jsp	

	on				
/shop/index	org.ibatis.jpetstore.presentation.action.IndexAction			/jsp/index.jsp	
/shop/listOrders	org.ibatis.jpetstore.presentation.action.ListOrdersAction	accountForm		/jsp/ListOrders.jsp	
/shop/newAccount	org.ibatis.jpetstore.presentation.action.NewAccountAction	workingAccountForm	/shop/index.do (org.ibatis.jpetstore.presentation.action.IndexAction)		/jsp/NewAccountForm.jsp
/shop/newAccountForm	org.ibatis.jpetstore.presentation.action.NewAccountFormAction	workingAccountForm		/jsp/NewAccountForm.jsp	
/shop/newOrder	org.ibatis.jpetstore.presentation.action.NewOrder	workingOrderForm		/jsp/ConfirmOrder.jsp /jsp/Shippin	/jsp/NewOrderForm.jsp

	Action			gForm.jsp /jsp/ViewOrder.jsp	
/shop/newOrderForm	org.ibatis.jpetstore.presentation.action.NewOrderFormAction	workingOrderForm		/jsp/NewOrderForm.jsp	
/shop/removeItemFromCart	org.ibatis.jpetstore.presentation.action.RemoveItemFromCartAction	cartForm		/jsp/Cart.jsp	
/shop/searchProducts	org.ibatis.jpetstore.presentation.action.SearchProductsAction	emptyForm		/jsp/SearchProducts.jsp	
/shop/signoff	org.ibatis.jpetstore.presentation.action.SignoffAction		/shop/index.do (org.ibatis.jpetstore.presentation.action.IndexAction)		
/shop/signon	org.ibatis.jpetstore.prese	accountFor	/shop/index.do		

	ntation.actio n.SignonAct ion	m	(org.ibatis.jp etstore.prese ntation.actio n.IndexActi on)		
/shop/signon Form	org.ibatis.jp etstore.prese ntation.actio n.DoNothin gAction			/jsp/SignonF orm.jsp	
/shop/update CartQuantiti es	org.ibatis.jp etstore.prese ntation.actio n.UpdateCar tQuantitiesA ction	cartForm		/jsp/Cart.jsp	
/shop/viewC art	org.ibatis.jp etstore.prese ntation.actio n.DoNothin gAction	cartForm		/jsp/Cart.jsp	
/shop/viewC ategory	org.ibatis.jp etstore.prese ntation.actio n.ViewCate goryAction	categoryFor m		/jsp/Categor y.jsp	/jsp/index.js p
/shop/viewIt em	org.ibatis.jp etstore.prese	itemForm		/jsp/Item.jsp	/jsp/Product. jsp

	ntation.actio n.ViewItem Action				
/shop/viewO rder	org.ibatis.jp etstore.prese ntation.actio n.ViewOrde rAction	accountFor m		/jsp/ViewOr der.jsp	
/shop/viewP roduct		productFor m		/jsp/Product. jsp	/jsp/index.js p

Table D-1-1 Relationships in Struts Configuration File

D.2 Relationships in iBATIS Configuration Files

Only one configuration file is analysed here. The others have the similar structure and can be processed in the same way.

D.2.1 iBATIS Configuration Files

```
<sql-map name="Item">

  <result-map name="light-result" class="org.ibatis.jpetstore.domain.Item">
    <property name="itemId" column="ITEMID" />
    <property name="listPrice" column="LISTPRICE" />
    <property name="unitCost" column="UNITCOST" />
    <property name="supplierId" column="SUPPLIER" />
    <property name="productId" column="PRODUCTID" />
    <property name="status" column="STATUS" />
    <property name="attribute1" column="ATTR1" />
    <property name="attribute2" column="ATTR2" />
    <property name="attribute3" column="ATTR3" />
    <property name="attribute4" column="ATTR4" />
    <property name="attribute5" column="ATTR5" />
  </result-map>

  <result-map name="result" class="org.ibatis.jpetstore.domain.Item">
    <property name="itemId" column="ITEMID" />
    <property name="listPrice" column="LISTPRICE" />
    <property name="unitCost" column="UNITCOST" />
    <property name="supplierId" column="SUPPLIER" />
    <property name="productId" column="PRODUCTID" />
```

```
  <property name="status" column="STATUS" />
  <property name="attribute1" column="ATTR1" />
  <property name="attribute2" column="ATTR2" />
  <property name="attribute3" column="ATTR3" />
  <property name="attribute4" column="ATTR4" />
  <property name="attribute5" column="ATTR5" />
  <property name="quantity" column="QTY" />
</result-map>

<mapped-statement name="getItem" inline-parameters="true" result-map="result">
  select * from ITEM item, INVENTORY inv
  where item.ITEMID = inv.ITEMID
    and item.ITEMID = #key#
  </mapped-statement>

  <mapped-statement name="getItemListByProduct" inline-parameters="true" result-map="light-
result">
    select * from ITEM where
    PRODUCTID = #key#
  </mapped-statement>

  <mapped-statement name="getItemList" result-map="light-result">
    select * from ITEM
  </mapped-statement>


  <mapped-statement name="updateInventoryQuantity" inline-parameters="true">
    update INVENTORY set
    QTY = #quantity#
    where ITEMID = #itemId#
  </mapped-statement>

</sql-map>
```

Listing D-2-1 Item.xml

D.2.2 Relationships in iBATIS Configuration File

Name
getItem
getItemListByProduct
getItemList
updateInventoryQuantity

Table to be continued on next page

Parameter Class	Parameter Map	Result Class	Result Map	Related Table
			org.ibatis.jpetstore.domain.Item	INVENTORY
				ITEM
			org.ibatis.jpetstore.domain.Item	ITEM
			org.ibatis.jpetstore.domain.Item	ITEM
				INVENTORY

Table to be continued on next page

Table D-2-1 Relationships in iBatis Configuration File

Related Class
org.ibatis.jpetstore.persistence.dao.ItemDao
org.ibatis.jpetstore.persistence.dao.cache.PetStoreCacheDao
org.ibatis.jpetstore.persistence.dao.map.ItemMapDao
org.ibatis.jpetstore.persistence.dao.ItemDao
org.ibatis.jpetstore.persistence.dao.cache.PetStoreCacheDao
org.ibatis.jpetstore.persistence.dao.map.ItemMapDao
org.ibatis.jpetstore.presentation.action.ViewProductAction
org.ibatis.jpetstore.persistence.dao.ItemDao
org.ibatis.jpetstore.persistence.dao.cache.PetStoreCacheDao
org.ibatis.jpetstore.persistence.dao.map.ItemMapDao
org.ibatis.jpetstore.presentation.form.ProductForm
org.ibatis.jpetstore.persistence.dao.InventoryDao
org.ibatis.jpetstore.persistence.dao.cache.PetStoreCacheDao
org.ibatis.jpetstore.persistence.dao.map.InventoryMapDao
org.ibatis.jpetstore.presentation.action.NewOrderAction

APPENDIX E Publications

[1] B. Qiao, H. Yang and A. O'Callaghan, "A Unified Software Reengineering Approach towards Model Driven Architecture Environment", Book Chapter in Software Evolution with UML and XML, Idea Group Publishing, ISBN: 1591404630, April 2005.

[2] W. C. Chu, C. Chang, C. Lu, H. Yang, H. Jiau, Y. Chung and B. Qiao, "Enhancing Software Maintainability by Unifying and Integrating Standards," Book Chapter in Advances in Software Maintenance Management: Technologies and Solutions, Idea Group Publishing, ISBN: 1591400856, January 2003.

[3] B. Qiao and H. Yang, "A Unified Solution to Architecture Recovery and Migration," De Montfort University Graduate Conference, April 2003.

[4] B. Qiao, Hongji Yang, William C. Chu, Baowen Xu, "Bridging Legacy Systems to Model Driven Architecture," In the Proceedings of The 27th IEEE Conference on Computer Software and Application, USA, 2003.

[5] S. Li, B. Qiao, Hongji Yang, H. Liao and H. Zhou, "System Quality Propagation in Reverse Architecturing," In the Proceedings of The 7th Integrated Design and Process Technology, 2003.

[6] B. Qiao and H. Yang, "Evolution of Web-Based System: An Architecture Centred Approach," Conference on Automation and Computer Science UK (CACSUK'02), Manchester, UK, Sept. 2002.